The Development of Secure Multi-Agent Systems

PhD Thesis

Helge T. Janicke

This thesis is submitted in partial fulfilment of the requirements for the Doctor of Philosophy, awarded by De Montfort University.

February 2007

Abstract

Security requirements must be addressed early and throughout the development of the system. They cannot be left to a late stage in the development process, as otherwise functional design-decisions may undermine security requirements. Honouring this principle we developed the SANTA framework which integrates the specification of *security*, *functional* and *temporal* requirements of Multi-Agent Systems (MAS) within a unifying and formal framework.

The specification and implementation of MAS is supported by the SANTA Wide-Spectrum Language (SANTA-WSL). SANTA-WSL allows for the expression of specifications and their implementations within the same language. The specification-oriented semantics of SANTA-WSL is given in Interval Temporal Logic (ITL), that is the formal foundation of all components in the framework. A formal foundation is key to the certification of MAS deployed in *security critical environments* where a breach in security may result in serious harm to people, equipment or missions. A SANTA-WSL specification comprises *agents*, *objects*, *policies* and *enforcement mechanisms*. Agents are active entities in the system; objects represent passive resources; policies express security requirements; and enforcement mechanisms define the effect of a policy on the execution. Policies can change dynamically according to time or events and can express history-dependent constraints. They are compositional, that is policies can be composed out of small, easier to comprehend components along a *temporal* and *structural* axis.

We show the advantages of policy composition for the specification and enforcement of policies. Compositionality is also important for verification, as properties of the overall policy can be inferred from the properties of its components. We show how abstract policy and enforcement specifications can be refined into concrete and implementable enforcement code that guarantees the compliance with original specification. On one hand policies depend on the history of the system's execution, on the other hand the execution of the system depend on the outcome of policy decisions. By addressing security, functional and temporal requirements in a unifying framework we are able to formalise and study their interactions.

Declaration

I declare that the work described in this thesis is original work undertaken by me between August 2003 and October 2006 for the degree of Doctor of Philosophy, at the Software Technology Research Laboratory (STRL), Department of Computer Science and Engineering (CSE), De Montfort University, United Kingdom.

Publications

1. Helge Janicke, Antonio Cau, François Siewe, Hussein Zedan and Kevin Jones. *A Compositional Event & Time-based Policy Model.* In Proceedings of the 7th International Workshop on Policies for Distributed Systems and Networks, London, Ontario, Canada, 2006.

2. Helge Janicke, François Siewe, Kevin Jones, Antonio Cau and Hussein Zedan. *Analysis and Run-time Verification of Dynamic Policies.* In Robert Ghanea-Hercock, Mark Greaves, Nick Jennings and Simon Thompson editors, Proceedings of the 1st Workshop on Defence Applications for Multi-Agent Systems (DAMAS'05), at AAMAS'05, Utrecht, Netherlands, 2005.

3. P. Beautement, D. Allsopp, M. Greaves, S. Goldsmith, S. Spires, S. Thompson and H. Janicke. *Autonomous Agents and Multi-Agent Systems (AAMAS) for the Military — Issues and Challenges.* In Robert Ghanea-Hercock, Mark Greaves, Nick Jennings and Simon Thompson editors, Proceedings of the 1st Workshop on Defence Applications for Multi-Agent Systems (DAMAS'05), at AAMAS'05, Utrecht, Netherlands, 2005.

4. Helge Janicke. *Securing Agents Systems with Dynamic Policies.* Student Paper. In the 7th Agent System Summer School (EASSS'05), Utrecht, Netherlands, 2005.

5. François Siewe, Helge Janicke and Kevin Jones. *Dynamic Access Control Policies and Web-Service Composition.* In Proceedings of the 1st Young Researcher Workshop on Service-Oriented Computing, Leicester, UK, 2005.

Acknowledgements

I would like to thank my supervisors Dr. Antonio Cau and Prof. Hussein Zedan for their guidance and support during my studies — especially for their patience during the many discussions that lead to this thesis. Many thanks also to the other members of STRL and the members of the DIF-DTC Agent Theme whose critical comments were most helpful.

My special gratitude is to my wonderful parents and family for supporting me all my life with their love and encouragement to continue my studies — I could not have done without them. I would especially like to thank my wife, for marrying me, supporting me, cheering me up and simply being there to fill my life with love.

The work was funded by the Data & Information Fusion - Defence Technology Center (DIF-DTC) through project 12.5.1, "Secure and Trusted Agents for Information Fusion".

Contents

1 Introduction **23**

2 Related Work **29**

2.1 Introduction 30
2.2 Agent and Multi-Agent Systems 30
2.2.1 Agents 33
2.2.2 Categorisation of Agents 44
2.2.3 Agents in the Context of this Work 54
2.3 Security 55
2.3.1 Access Control Models 56
2.3.2 Policy Languages 64
2.3.3 Enforcement 74
2.4 Summary 76

3 Computational Model **77**

3.1 Introduction 78
3.2 Agents 80
3.2.1 Agent State 81
3.2.2 Agent Capabilities 81
3.2.3 Single Agent Execution Model 83
3.2.4 Agents in SANTA 85
3.3 Objects 85
3.3.1 Object State 86
3.3.2 Interfaces 86
3.3.3 Objects in SANTA 87
3.4 Policies 87
3.4.1 Authorisation 88
3.4.2 Delegation 88

3.4.3 Obligation . . . 89
3.4.4 Integrity . . . 89
3.4.5 Policies in SANTA . . . 90
3.5 Enforcement Mechanisms . . . 90
3.5.1 Vigilant Agents . . . 91
3.5.2 Vigilant Objects . . . 91
3.5.3 Security Enforcer . . . 91
3.5.4 Enforcement Mechanism in SANTA . . . 92
3.6 Summary . . . 92

4 Preliminaries **95**
4.1 Introduction . . . 96
4.2 Syntax and Semantics . . . 96
4.3 Formal Semantics . . . 98
4.4 Derived constructs . . . 99
4.5 Operator Always-followed-by . . . 101
4.6 Temporal Projection . . . 101

5 Agents **103**
5.1 Introduction . . . 104
5.2 Agents in SANTA . . . 104
5.2.1 Agent State Variables . . . 105
5.2.2 Agent Capabilities . . . 107
5.2.3 Agent Deliberation . . . 110
5.2.4 Motivating Example . . . 111
5.3 Single Agent Semantics . . . 112
5.3.1 Initialisation Phase . . . 113
5.3.2 Deliberation Phase . . . 113
5.3.3 Statement Semantics . . . 115
5.3.4 Enforcement Phase . . . 117
5.3.5 Execution Phase . . . 118
5.3.6 Termination Phase . . . 122
5.3.7 Agent Semantics . . . 123
5.4 Multi-Agent System Semantics . . . 123
5.5 Example of $SMAS_1$ . . . 124
5.6 Summary . . . 127

6 Objects **129**
6.1 Introduction . . . 130
6.2 Objects in SANTA . . . 130
6.2.1 Object State Variables . . . 131
6.2.2 Object Interfaces . . . 131
6.2.3 Remote Actions . . . 131
6.2.4 Motivating Example . . . 133
6.3 Single Agent Semantics . . . 134
6.3.1 Execution Phase . . . 134
6.4 Single Object Semantics . . . 141
6.4.1 Object Initialisation . . . 141
6.4.2 Interface Invocation . . . 142
6.5 Multi Agent Semantics . . . 144
6.6 Example of $SMAS_2$ . . . 144
6.7 Summary . . . 147

7 Policies **149**
7.1 Introduction . . . 150
7.2 Policies in SANTA . . . 153
7.2.1 Policy Rules . . . 155
7.2.2 Authorisation Rules . . . 160
7.2.3 Delegation Rules . . . 163
7.2.4 Obligation Rules . . . 165
7.2.5 Integrity Rules . . . 166
7.2.6 Simple Policies . . . 170
7.3 Semantics of Policy Rules and Simple Policies . . . 170
7.3.1 Control Variables . . . 170
7.3.2 Policy Scope and Free Variables . . . 172
7.3.3 Rules . . . 173
7.3.4 Simple Policies . . . 183
7.4 Policy Composition . . . 183
7.4.1 Temporal Composition . . . 184
7.4.2 Structural Composition . . . 187
7.5 Semantics of Policy Composition . . . 191
7.5.1 Temporal Composition . . . 191
7.5.2 Structural Composition . . . 191
7.6 Summary . . . 197

8 Enforcement 199
8.1 Introduction . . . 200
8.2 Enforcement Mechanisms in SANTA . . . 201
8.3 Subjects, Objects and Actions . . . 202
8.4 Complete Policy Specification . . . 202
8.5 Vigilant Agent . . . 204
8.5.1 Enforceable Policies . . . 205
8.5.2 Mapping Policy- and System States . . . 205
8.5.3 Enforcing Authorisation . . . 207
8.5.4 Enforcing Obligations . . . 208
8.5.5 Enforcing Integrity . . . 208
8.6 Vigilant Object . . . 209
8.6.1 Enforceable Policies . . . 209
8.6.2 Mapping Policy- and System States . . . 209
8.6.3 Enforcing Authorisation . . . 212
8.6.4 Enforcing Integrity . . . 213
8.7 Security Enforcer . . . 214
8.7.1 Enforceable Policies . . . 214
8.7.2 Motivating Example . . . 215
8.7.3 Semantics of the Security Enforcer . . . 216
8.7.4 Mapping Policy- and System States . . . 217
8.7.5 Enforcing Authorisation . . . 218
8.7.6 Enforcing Integrity . . . 218
8.8 Summary . . . 219

9 Refinement 221
9.1 Introduction . . . 222
9.2 Refinement Rules . . . 223
9.3 Refinement of the $SMAS_1$ Example . . . 226
9.3.1 Initialisation . . . 227
9.3.2 Deliberation . . . 230
9.3.3 Enforcement . . . 231
9.3.4 Execution . . . 231
9.4 Refining a Vigilant Agent . . . 233
9.4.1 Determining the Required Enforcement History . . . 237
9.4.2 Rule Transformation . . . 238
9.4.3 Determining the Chopping Points . . . 239

9.4.4 Transformation of the Vigilant Agent Example . . . 242
9.4.5 Refining the Enforcement Time . . . 243
9.4.6 Maintaining the History . . . 244
9.4.7 Refining Obligation Rules . . . 245
9.4.8 Refining Authorisations . . . 247
9.4.9 Refining Integrity Rules . . . 249
9.4.10 Automation . . . 250
9.4.11 Enforcing Composed Policies . . . 253
9.5 Summary . . . 254

10 Analysis **257**
10.1 Introduction . . . 258
10.2 SPAT Architecture . . . 258
10.3 Case Study . . . 261
10.3.1 Scenario . . . 261
10.4 Policy Specification . . . 262
10.5 Policy Validation . . . 265
10.5.1 Scenario Visualisation . . . 266
10.5.2 Policy Decisions . . . 267
10.5.3 Graphical Representation . . . 268
10.5.4 Explanation Component . . . 270
10.6 Summary . . . 271

11 Conclusion **273**
11.1 Introduction . . . 274
11.2 Summary . . . 274
11.3 Contributions . . . 276
11.3.1 Integration of Functional and Security Requirements . . . 276
11.3.2 Extensions to the original Policy Language . . . 277
11.3.3 Policy as an Abstraction . . . 279
11.3.4 Enforcement . . . 280
11.4 Critical Remarks . . . 281
11.5 Future Work . . . 284

Glossary **301**

List of Figures

1.1 Outline of the Proposed Framework 26

2.1 Relations between Properties 36
2.2 Agent-Taxonomy given by Franklin & Graesser 44
2.3 A Part View of an Agent Topology (adopted from Nwana) 45
2.4 Interface Agents (adapted from Maes [91]) 47
2.5 Network of agent platforms, mobile agents 49
2.6 Comparison between both approaches, adopted from Brooks [36] 52
2.7 Hybrid Architectures: TouringMachines 53
2.8 Hybrid Architectures: INTERRAP 54
2.9 Access-Control Matrix and Access-Control List 58
2.10 Lattice of security labels and compartments 60
2.11 Abstraction from user to roles 62
2.12 Role-Hierarchies 64
2.13 Bell-Lapadula's simple security condition in LaSCO 67
2.14 Policy Enforcement Model 74
2.15 No *Send* after *Read* 75

3.1 General Overview of secure MAS components 79
3.2 Phases in the Single Agent Execution Model 83

4.1 Informal Semantics of $f_1 ; f_2$ 97
4.2 Informal Semantics of f^* 97
4.3 Informal Semantics of $f \mapsto w$ 101
4.4 Example of Temporal Projection 102

5.1 Specification level behaviour of the $SMAS_1$ example 124
5.2 Specification level behaviour of action `inc` 126

6.1 Schematic of remote action statement 136

6.2 Abstract behaviour of the $SMAS_2$ example 146

7.1 Informal Interpretation of a Policy Rule 157
7.2 1. Integrity Rule: No previous access to **get** 169
7.3 2. Integrity Rule: One previous access to **get** 169
7.4 3. Integrity Rule: Two or more previous accesses to **get** 169
7.5 Operator Always-Followed-By 173
7.6 Primed Variables in Integrity Rules 177
7.7 Example of referencing parameter values in the past 182

8.1 Abstraction Levels 201
8.2 State projection for vigilant Agent 206
8.3 Vigilant Agent enforcing a policy 207
8.4 Vigilant Object enforcing a policy 212
8.5 Abstraction Level for Environmental Integrity Policies 213

9.1 Specification level behaviour of the $SMAS_1$ example 227
9.2 Specification level behaviour of action `inc` 227
9.3 Implementation Level behaviour of the initialisation phase 229
9.4 Enforcement Time 244
9.5 Example of Automated Rule Analysis 251
9.6 Example of Automated Rule Analysis 252

10.1 SPAT Architecture 259
10.2 Platoon navigating a terrain 262
10.3 Prototype of the Policy Specification Module 265
10.4 Tabular View of the SPAT Analysis Module 268
10.5 Graphical View 269
10.6 Explanation Component 270

11.1 Outline of the Trust Management Framework 286

List of Tables

2.1 Properties of agents, adopted from [60] 36

4.1 Syntax of ITL 97

5.1 Accessibility of Agent State Variables 106
5.2 States in the Execution of Listing 5.3 127

6.1 Phases in Remote Action Execution 138
6.2 Execution of the $SMAS_2$ Example. 147

7.1 Consequences in Policy Rules 160
7.2 Operators for the temporal composition of policies. 191
7.3 Referencing local decisions 197

9.1 States in the Initialisation Phase 229
9.2 States in the Deliberation and Enforcement Phase 230
9.3 States in the Execution Phase 233
9.4 Vigilantly Enforced Policy 236

Listings

3.1 Agent Definition in SANTA 85
3.2 Object Definition in SANTA 87
3.3 Policy Definition in SANTA 90
3.4 Examples of Enforcement Mechanisms 92
5.1 EBNF for $SMAS_1$ 104
5.2 EBNF for Expressions 110
5.3 Example Agent Specification 112
5.4 Example Agent Specification 124
6.1 EBNF for $SMAS_2$ 130
6.2 Example for explicitly named remote actions 133
6.3 Motivating Example for $SMAS_2$ 133
6.4 Motivating Example for $SMAS_2$ 145
7.1 EBNF for Policy Syntax 154
7.2 Example Positive Authorisation Rules 161
7.3 Example Negative Authorisation Rules 161
7.4 Example Decision Rules 162
7.5 Delegation and Revocation 163
7.6 Example Delegation Rules 164
7.7 Effects of delegations 164
7.8 Example Obligation Rule 165
7.9 Example Integrity Rule 167
7.10 Example Integrity Rule 167
7.11 Example Integrity Rule 168
8.1 Motivating Example for the Security Enforcer 215

Chapter 1

Introduction

Motivation

With the growth of the World Wide Web (WWW) more and more information becomes available, in form of scientific articles, results of experiments, news and other on-line services. But it is not only the part of the internet that we can access through web-pages that grows at such an enormous rate, increasingly businesses and governmental institutions use the internet, or similar networks, to communicate and share (confidential) information in order to be competitive or to provide better services.

The amount of information does lead to an information overload, viz. the problem of determining which information is relevant and which is not. This is not only a difficulty for private users, but especially for global businesses and governmental institutions that use and share information with their partners. Sharing information efficiently and securely is especially important in the military domain, where increased situational awareness and shorter Observe, Orientate, Decide, Act (OODA) cycles are critical success factors. Dedicated decision support systems are devised to aid in the analysis of information and provide support for human decision making under severe time constraints whilst maintaining a high level of assurance in the security of the system.

We identified in [19] requirements and problems that information and decision support systems face in the military domain, that – by its nature – is characterised by uncertainty and disruption. Based on the identified needs we concluded that Agents and Multi Agent System technologies are one of the best candidates to provide these systems. However, with the increased level of awareness and the higher availability of mission critical information, security becomes a major concern.

Problem Statement

Security cannot be bolt on. It must be addressed together with the functional and temporal requirements early and throughout the system's development life-cycle. If security requirements are not addressed in the early stages of the development process, the system design is likely to prevent their implementation — leading either to insecure implementations or to a complete (and expensive) redesign of the affected components.

Especially for *security critical applications*, viz. applications in which the violation of the system's security requirements can cause actual bodily harm or loss of life, it is paramount that mechanisms that ensure the compliance with the security requirements are in place and can be certified. By certification we mean that a high level of assurance can be provided that the implementation of the security mechanisms ensure that none of the security requirements can possibly be violated.

The key advantage of Multi Agent Systems for information and decision support systems is the unsupervised collaboration of autonomous agents to achieve an adopted goal. Collaboration, however, requires the exchange of potentially security critical information. To guarantee that security requirements are not violated it is important to limit the autonomy of the agents in the Multi Agent System and their access to system resources.

In the context of Multi Agent Systems the employed mechanisms must be *flexible*, viz. the enforced security requirements can change dynamically on time or events, and *modular*, viz. the mechanisms should be loosely coupled with the agent design and should not limit the architecture that is used for the implementation of individual agents. If the security mechanisms are too static and impose too many constraints on the implementation of system components, they destroy the very advantages that Agent and Multi Agent System technology has to offer for applications used in uncertain and disruptive environments.

Approach to Solution

To be able to provide such a high level of assurance, we integrate the specification of *security*, *functional* and *temporal* requirements within a unifying and formal framework. This allows for the analysis of the effect that the enforcement of security requirements has on the behaviour of the Multi Agent System, which is the main goal of any certification process.

The use of formal methods, viz. the creation of a mathematical model of the system that accurately describes its functioning, is a key factor, as it allows for the unambiguous specification of the requirements and their verification using mathematical machinery that has been developed and improved upon over years. The use of mathematical formulations to express security requirements, such as access control or integrity constraints, has been

identified as a corner stone for the certification of military information systems in e.g. [85].

Today, policy languages are used to formulate security requirements at a level that is more accessible to administrators who will inevitably maintain the systems. They provide an abstraction from the concrete models that are encoded in the system. The idea is to separate the policies from their implementations and to introduce management components in the system that interpret and *enforce* these policies [126]. The advantage of this modular approach is that it permits the timely adaptation to changes in the system and allows to react quickly to formerly undiscovered vulnerabilities. This flexibility is essential in military and business environments.

Although many policy languages have been proposed (e.g. [72, 108, 134, 118, 46, 20]) their ability to express dynamic change of policies based on time or the occurrence of events is limited. Dynamically changing policies have been recently addressed in [124] by policy composition at the specification level. We believe that dynamically changing policies play an important role for the specification of security requirements for Multi Agent Systems in the before mentioned domains, as they allow for the dynamic adoption of a different security policy based on the current situation.

We use the model proposed in [124] as the foundation of our work as it also provides a compositional approach to policy specification and verification. We feel that the compositionality of the model improves the scalability of the formal approach. Using this model, policies can be specified as individual components, for example policies that apply in different situations are specified individually, and are then combined to form the overall policy of the system. Similarly, the compositionality of the model does allow for the compositional verification of policies, viz. properties of the system policy can be inferred from properties of its components.

Another important aspect of the policy model is that it allows for the specification of history-based policies [3]. For example the requirement "if a user at some point in the past *read* a file that is *secret* then the same user cannot *write* to a file that is *unclassified*" can be expressed naturally in the model. This allows for the expression of a large class of security requirements, including complex policies such as the Chinese Wall Policy [35].

By providing a unifying and formal framework, named SANTA, we are able to integrate the functional specification of the system with its security and temporal requirements. This allows us to study the effect that the enforcement of dynamically changing security policies has on the behaviour of the Multi Agent System. Through the use of Interval Temporal Logic (ITL), we are able to study the timeliness of the policy enforcement and show that the system-implementation satisfies the functional, security and temporal requirements. Figure 1.1 outlines the proposed framework.

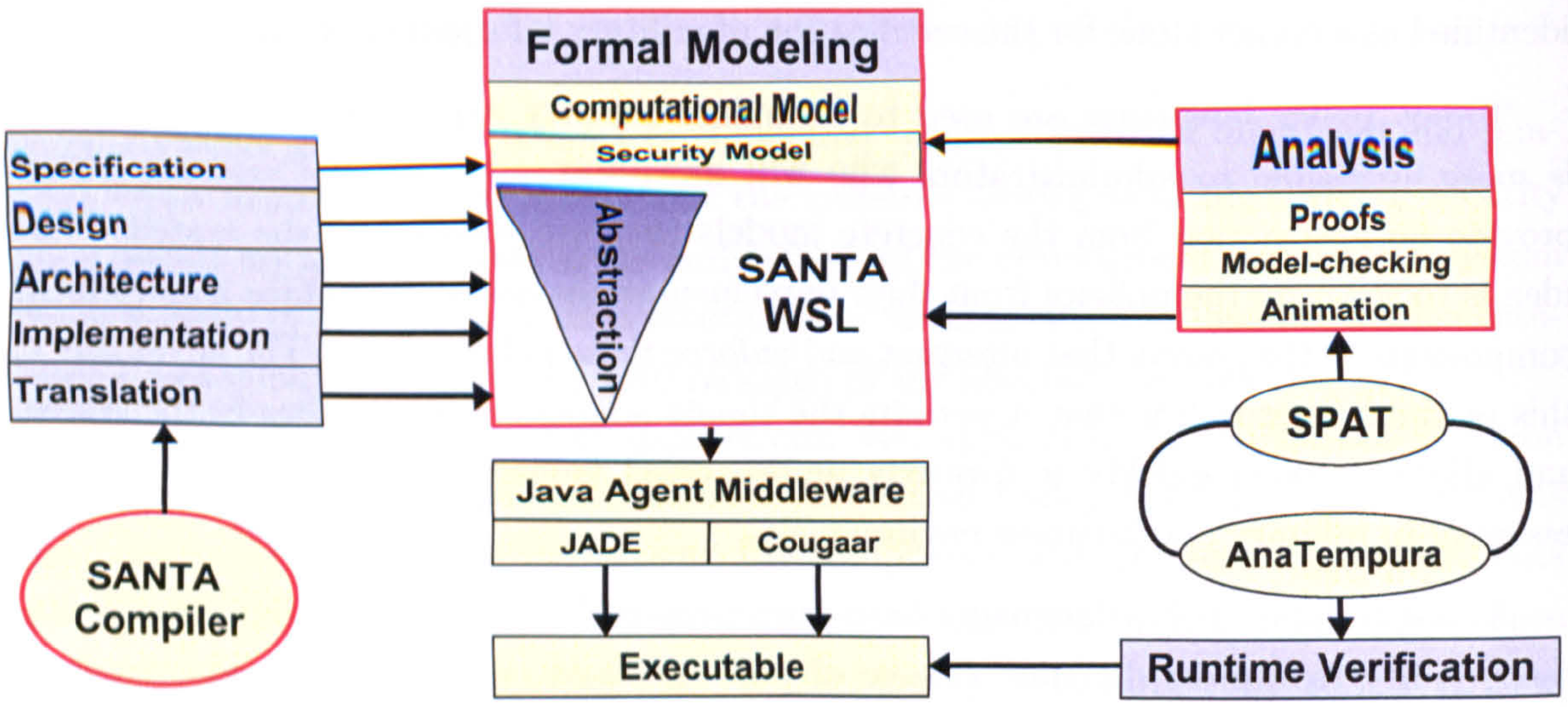


Figure 1.1: Outline of the Proposed Framework

The SANTA framework is organised along a theoretical – to – applied axis. The foundation is a formal computational model and a compatible security model. To aid in the specification and design of the system we provide linguistic support, in form of the SANTA Wide-Spectrum Language (SANTA-WSL), for the specification of a Secure Multi Agent System (SMAS). The language is agent-based and provides constructs for the specification of reactive agents, objects that represent shared resources, security policies and their enforcement mechanisms. SANTA-WSL is a *wide-spectrum language*, viz. the language allows for specification constructs and concrete implementations to coexist within the same program. The specification-oriented semantics of SANTA-WSL is given in ITL that is the underlying logical foundation of the SANTA framework.

The development of Secure Multi Agent Systems using SANTA-WSL is based on the notion of *refinement*, that is the stepwise transformation of a specification into a concrete and implementable program through the application of sound, semantic preserving refinement rules. Policies are defined at a higher level of abstraction than the system implementation itself. For example an authorisation policy defines the conditions under which a user can access a specific resource in a declarative manner — without detailing *how* this constraint is actually implemented. Enforcement mechanisms are also defined at a high abstraction level. They state properties that the system must satisfy to comply with a specific policy. During the development process policies and enforcement mechanisms are refined into concrete and deterministic enforcement code.

By refining the policy and their enforcement mechanisms into concrete, provably correct enforcement code, we are able to provide high assurance in the system's security mechanisms. This is important for the certification of Secure Multi Agent Systems de-

veloped using the SANTA framework, that would otherwise have to rely on review and validation techniques.

The concrete SANTA programs should be readily implementable (or even compilable) in agent-frameworks such as JADE [129], although this is not the main focus of the thesis. Part of the SANTA framework forms also the tool-support for analysis, where we especially focus on the early prototyping of policies, their animation and analysis using the Security Policy Analysis Tool (SPAT).

Outline

The thesis is structured as follows. In Chapter 2 we critically review related work in the areas of Multi Agent Systems and policy specification/enforcement. In Chapter 3 we provide an introduction to the SANTA framework and SANTA-WSL that serves as an overview for the formalisation in the subsequent chapters. For completeness of the presented work, we introduce in Chapter 4 the syntax and semantics of ITL. Readers who are already familiar with ITL may wish to skip this chapter.

To make the formalisation of the framework more accessible, we gradually extend SANTA-WSL from Chapter 5 to 8. In Chapter 5 we concern ourselves with the formal specification of agents. We refer to the subset that is introduced in this chapter as $SMAS_1$, viz. a $SMAS_1$ is a SMAS that contains *only* agent specifications. We extend the specification of a $SMAS_1$ by introducing objects in Chapter 6 and formalise the communication between agents and objects. We refer to this subset as $SMAS_2$, viz. a $SMAS_2$ is a SMAS that contains only agents and objects.

In Chapter 7 we present the policy model presented in [124] and extensions for the specification of obligation and integrity requirements as well as scoping and structural composition [74]. We also provide examples of well-known policies such as Bell LaPadula [21] and Chinese Wall [35]. In Chapter 8 we then define (de-)centralised enforcement mechanisms that link policies with agents and objects. We refer to a SMAS as a $SMAS_3$ to explicitly state that the SMAS contains agents, objects, policies and enforcement mechanisms.

In Chapter 9 we define formally what we mean by refinement and provide refinement rules that are used subsequently to show how agents in a SMAS can be developed. We also apply the refinement rules to show how enforcement mechanisms and policies can be refined into concrete enforcement code.

Chapter 10 describes the design of SPAT that has been implemented as a prototype. The tool is used for the early prototyping and validation of dynamically changing policies. It is intended to be used by policy designers to validate that a policy specification does yield

the desired effect before the policy is deployed. We summarise the thesis in Chapter 11 where we emphasise the contributions of this work, and review them critically. The chapter also outlines future work to improve the SANTA framework.

Chapter 2

Related Work

In this chapter we introduce the concept of Software Agents and Multi-Agent Systems and review related work in the area of policy languages that can be enforced to control their execution.

2.1 Introduction

This chapter is divided into two parts. The first part, Section 2.2, provides an introduction to agent-based systems. We survey the different view-points on this topic that have influenced research in this area over the years. The aim is to provide an overview of what agent-based systems are and the properties that are commonly associated with agents. We then point out the key properties of agents that are addressed in this thesis.

The second part, Section 2.3, critically reviews related work in the area of security with the emphasis on policy-based languages and control mechanisms for distributed systems in general and for Multi Agent Systems in particular. Following the review on policy models and languages, we discuss the enforcement of policies. The chapter concludes with a short summary in Section 2.4.

2.2 Agent and Multi-Agent Systems

The idea of agents is old — mankind has always dreamed of creating artificial intelligence and artificial life-forms, that would help in our day to day business. The idea of agents is indeed about learning and assisting human users. According to Allan Kay [81] the idea of software agents attributes to John McCarthy and Oliver G. Selfridge during their work in the 1950s at Massachusetts Institute of Technology (MIT). Kay describes their vision as follows:

> *They had in view a system that, when given a goal could carry out the details of the appropriate computer operations and could ask back for and receive advice, offered in human terms, when it was stuck. An agent would be a soft robot living and doing its business within the computer world. [81] (cited by [33])*

An agent that could deduce execution plans from a given goal to *carry out appropriate computer operations* itself and that accepts instructions *offered in human terms* immediately leads to the research field of Artificial Intelligence (AI), which had and still has the main influence on todays research on agents. Jennings et. al. [77] state that AI *is all about creating intelligent artifacts* and that if these artifacts can *sense and act in some environment* they can be considered to be agents, indeed. Some specialised groups within the AI community are more closely related to agents. Namely these are Artificial Intelligence Planning (AIP) and Distributed Artificial Intelligence (DAI). Research in these areas was categorised by Nwana [107] as being the first strand in the history of agent development.

Artificial Intelligence Planning is usually associated with the Stanford Research Institute Problem Solver (STRIPS) [55] planning system which uses symbolic representations

of the agent's environment, its capabilities and goals to describe the system. The planning algorithm employs symbolic reasoning techniques to create a plan that can be executed by the agent in order to achieve its goal. In the 1980s it became clear that although the planning algorithms in STRIPS and its descendants provide reasonable performance they do not scale well when applied to more realistic, real world examples. Rodney Brooks has been one of the front runners in questioning the applicability of symbolic reasoning to real world examples in AI. Based on his experience he proposed an alternative to symbolic AI, variously known as behavioural AI, reactive AI or situated AI [37]. We discuss the resulting *subsumption architecture* [36] later in section 2.2.2.

Distributed Artificial Intelligence itself can be divided into two separate sub-domains 1.) Distributed Problem Solving (DPS) and 2.) Multi Agent System (MAS). The former concentrates on how knowledge can be distributed between different modules that in turn cooperatively solve the problem. In Distributed Problem Solving systems the communication and interaction strategies are usually predefined and form an integral part of the system. On the contrary the agents in Multi Agent System are autonomous problem solvers, which are loosely coupled and aimed to solve problems that would otherwise be beyond the capability of each single agent. Although Multi Agent System originated from Artificial Intelligence the word is today often used in a different sense: to denote systems that can be seen as they are constituted out of several agents [77].

The second strand of agent development, that has been identified by Nwana [107] started in the 1990s with the *diversification in the types of agents.* This was the time when the term agent has been picked up by researchers from a variety of different fields such as robotics, artificial life, distributed object computing, human-computer interaction — to enumerate only a few. Since then the already not well defined notion of *what an agent is* has become more vague and the categorisation of agent types even more complex. Having in view the now much broader area of agent research Nwana sees the Actor model proposed by Agha and Hewitt as one of the first models to describe Multi Agent Systems. In this model an actor is a self contained, interactive and continuously executing object. As discussed in further detail in the subsequent section the necessity of these three properties of agenthood, is widely accepted.

> *An actor is a computational agent which has an email address and a behaviour. Actors communicate by message passing and carry out their actions concurrently. [107]*

Undeniable other fields of research such as Object Oriented Programming (OOP), concurrent object-based systems as well as human-computer interface design, did contribute significantly to current agents research. Zambonelli et.al. state in [142] that for researchers

coming from the field of Object Oriented Programming agents are often seen as being (specialised) objects. Although it is true that most of the todays agent based applications are written in an Object Oriented Programming environment, agents are to be seen as a more abstract way of describing complex, highly interactive, distributed systems. One approach to develop Multi Agent System — the GAIA Methodology — is presented in [142]. It is based on the identification of organisations and roles, which later in the development are assigned to concrete agents.

The idea of using agents as an abstraction and directly create Multi Agent System by programming agents and their relationships was proposed by Shoham in the 1990s. His ideas were manifested in the Agent Orientated Programming (AOP) paradigm, that like Object Oriented Programming abstracts from the concrete implementation and describes entities (agents), their relationships and interaction. His work resulted in one of the first agent languages called AGENT0 in which an agent is essentially described in terms of *capabilities*, *initial beliefs* and *commitments* and so called *commitment rules* which are responsible for the actual action the agent does take. Since then other researchers invented agent-languages to describe Multi Agent System (e.g. [143, 58, 50, 92]).

Jennings et.al. [77] critically compare both paradigms and highlight the sometimes not obvious differences between agent oriented programming and object oriented programming. One of the most notable differences is the level of abstraction and compositionality that agents provide. Unlike an object, that encapsulates methods and attributes which can be accessed from other objects, an agent additionally encapsulates the control over its actions and state. This allows the pro-active behaviour that is often associated with agents [142].

Research in agents today can be seen as a joint venture of all kinds of different areas, not limited to Computer Science only. For example Philosophy and Psychology had and will have an important influence on todays agents. The philosopher Dennett coined the term *intentional systems*, by which he denotes systems in which the *behaviour* of entities *can be predicted by the method of attributing belief, desires and rational acumen.* A short list of attitudes emerging from this *intentional stance* is given in [140]. Out of these attitudes three form the basis of Rao & Georgeffs Belief, Desire, Intention (BDI) architecture [115], which is probably one of the best known architectures to characterise the mental state of agents.

The psychological aspect of agents is more obvious when *expert assistants* come into picture. In these systems a proactive user interface learns and assists the user in his work. Software Agents that assist the user of a (computer) system in her day to day tasks are becoming more and more important the more complex todays systems get. Nicholas Negroponte describes his view of agents in [106] as follows:

> *The best metaphor [...] for a human-computer interface is that of a well-trained English butler. The "agent" answers the phone recognizes the callers, disturbs you when appropriate, and may even tell a white lie on your behalf. [106]*

To be able to assist in a helpful, productive way the agent must possess knowledge about the user. Since this knowledge cannot be statically expressed for a specific program (as it is traditionally done), but needs to adapt to the changes of users experience and preferences, learning becomes of utmost importance for agents in the role of expert assistants. Pattie Maes emphasises that the agent

> *[...] does not act as an interface or layer between the user and the application. It rather behaves as a personal assistant which cooperates with the user on the task. The user is able to bypass the agent. [91]*

These cooperation and learning issues again draw the attention to the field of Artificial Intelligence. Natural language processing, speech recognition [15] and learning by demonstration [127] play an important role in the development of personal assistants.

Another important issue that is to be mentioned arises with the openness of Multi Agent System. Traditionally, Multi Agent System for Distributed Problem Solving were closed systems, in which all agents in the Multi Agent System are known and cooperate. With the second strand of agent development, and especially with the invention of *mobile* agents [42] that migrate from server to server through the internet, in order to accomplish tasks, cooperation of agents cannot be assumed. Beside standardisation [56] and communication issues [57], that are needed to be addressed in this heterogeneous environment, the question of how to deal with competitive agents and how to protect agents from being tempered or spoofed did arise. Despite the obvious need, security concerns have only fairly recently been addressed by the agent community (eg. [32, 132, 110, 34]).

This short review of the history of agent development should give the reader an idea, of how complex and multifaceted agents are. To summarise: The idea of agents started in the field of Artificial Intelligence and was picked up by various other disciplines in the early 1990s. Where AI traditionally addressed the learning, planning and collaboration aspect of agents various other aspects like mobility, fault tolerance and user interaction etc. came into picture. In the next section we discuss the different notions of *what agent is* and identify properties that are strongly associated with agenthood.

2.2.1 Agents

Unfortunately the field of agent research is vast and most of the definitions that are given to agent seem to originate from a specific application the author had in mind. Nwana states that there is

> *[...] as much chance of agreeing on a consensus definition for the word 'agent' as AI researchers have of arriving at one for 'artificial intelligence' itself — nil [107]*

Facing this, it seems to be obligatory for publications in the area of agents to start with a definition what the author exactly means by *agent*. This section firstly reviews some definitions of agents that were given by different authors and then describes the properties most commonly associated with agents.

Although Nwana deems a consensus definition of agency as infeasible and prefers to view the term agent as an umbrella term, she gives the following description of agency:

> *[An agent is] a component of software or hardware which is capable of acting exactly in order to accomplish tasks on behalf of its user. [107]*

This notion of agents is probably one of the broadest and seems to be very much oriented on the definition of the word agent, that is given in dictionaries. According to the Webster 1913 Dictionary [137] the word is derived from the Greek word *agere*, to *act*. It defines agent as

1. *One who exerts power, or has the power to act; an actor.*
2. *One who acts for, or in the place of, another, by authority from him; one intrusted with the business of another; a substitute; a deputy; a factor.*
3. *An active power or cause; that which has the power to produce an effect; as, a physical, chemical, or medicinal agent; as, heat is a powerful agent.*

The ability to act is the central property, that is associated with agents. In the above definitions, acting is still very general. Shoham gives a definition that is accepted in the AI community and according to Bradshaw [33] is also acceptable for a wider community. This definition is more concrete in describing the agent's ability to act.

> *Most often, when people in AI use the term 'agent,' they refer to an entity that functions continuously and autonomously in an environment in which other processes take place and other agents exist. [123]*

It emphasises on the continuity and autonomy of an agent and explicitly denotes the importance of the environment in which the agent is executing. Even stronger is the impact of the environment on the definition given by Franklin & Graesser:

> *An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future. [60]*

Here the emphasis lies on the autonomy and the relationship between the agent and its environment. Since this relationship is a crucial part of their definition, Franklin & Graesser state that *systems are agents or not with respect to some environment* [60]. This does reflect that, if the environment of the agent is changed, the agent might loose its status of being an agent w.r.t its environment. A similar definition is given by Wooldridge & Jennings.

> *An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives [139]*

This definition does also focus on the situatedness of an agent in its environment, its autonomy and goal-orientation. The list of definitions could be continued further, but the one mentioned should be sufficient to relate to the following properties that agents can exhibit. The following list of properties is adapted from Bradshaw:

Reactivity *The ability to selectively sense and act.*

Autonomy *Goal-directedness, proactive and self-starting behaviour.*

Collaborative behaviour *Can work in concert with other agents to achieve a common goal.*

"Knowledge Level" Communication ability *The ability to communicate with persons and other agents with language more resembling humanlike "speech acts" than typical symbol-level program-to-program protocols.*

Inferential capability *Can act on abstract task specification using prior knowledge of general goals and preferred methods to achieve flexibility; goes beyond the information given, and may have explicit models of self, user, situation, and/or agents.*

Temporal continuity *Persistence of identity and state over a long period of time.*

Personality *The capability of manifesting the attributes of a believable character such as emotion.*

Adaptivity *Being able to learn and improve with experience.*

Mobility *Being able to migrate in a self directed way from one host platform to another.*

[33]

These properties have also been identified by Franklin & Graesser and are reflected in their definition. For them the properties reactivity, autonomy, goal-orientation and temporal continuity are essential to agents, whereas the others are used to categorise different classes of agents. Table 2.1 shows the properties they have identified by reviewing several definitions of agents.

Properties used in the definition		
Property	*Other Names*	*Meaning*
reactive	sensing and acting	responds in a timely fashion to changes in the environment
autonomous		exercises control over its own actions
goal-oriented	pro-active, purposeful	does not simply act in response to the environment
temporally continuous		is a continuously running process
Properties used for further classification		
Property	*Other Names*	*Meaning*
communicative	socially able	communicates with other agents, perhaps including people
learning	adaptive	changes its behaviour based on its previous experience
mobile		able to transport itself from one machine to another
flexible		actions are not scripted
character		believable "personality" and emotional state

Table 2.1: Properties of agents, adopted from [60]

In most of the literature the definition of an agent is given by a enumeration of properties that the agent does exhibit. According to these properties classifications of different agent types have been proposed [107, 60]. It is remarkable to see how many different notions of what an agent is can be found in the literature, but even more fascinating is the divergence in the interpretation of these properties themselves. One example is *autonomy*. Although this property is the main constituent of most of the definitions it is not really clear what various researchers understand when using this term (Compare Table 2.1 and Bradshaw's list).

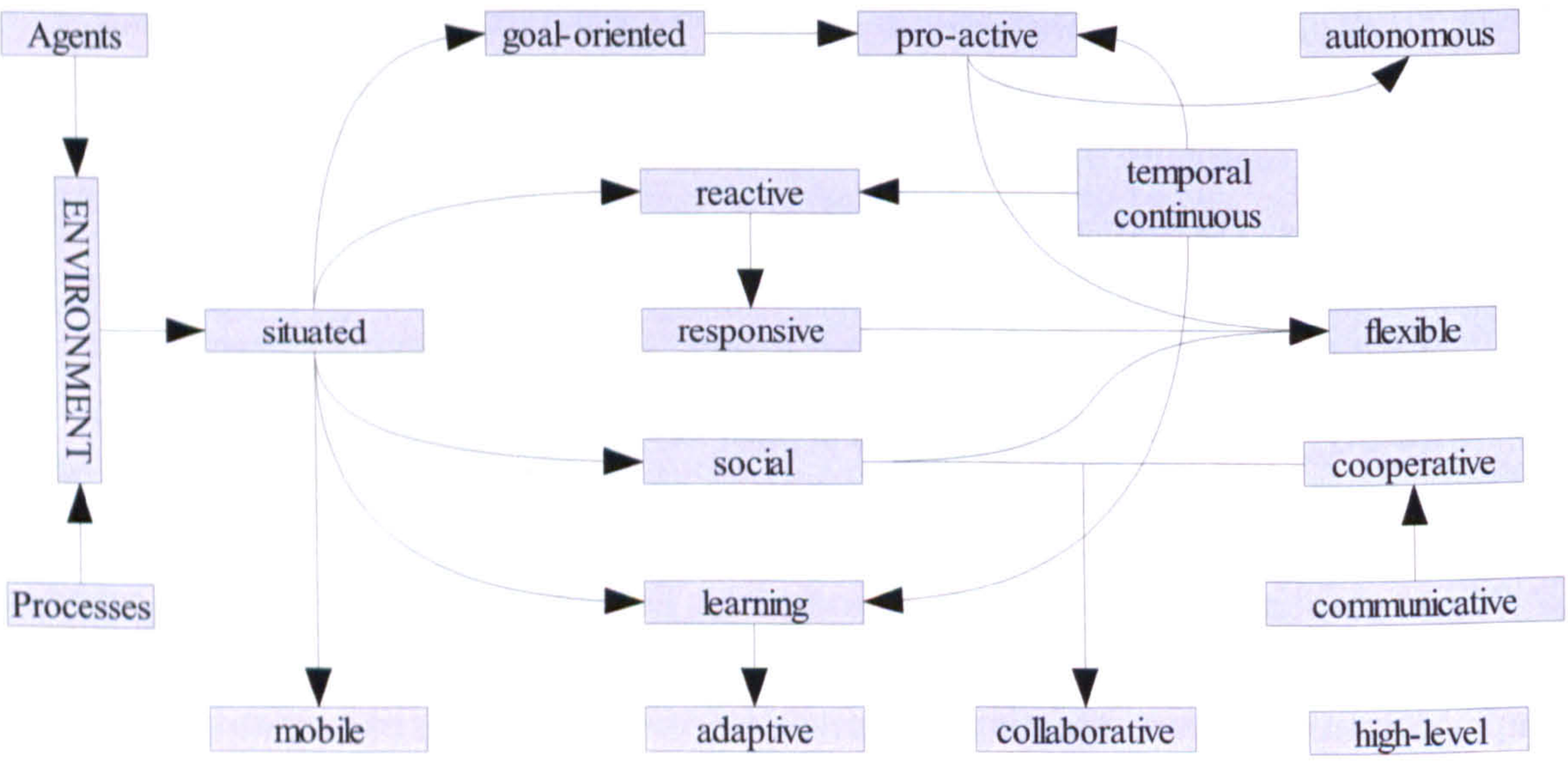


Figure 2.1: Relations between Properties

We will not try to find a solution for this problem, since this must be solved by the community as a whole, but show some relations between properties that might not be obvious. Figure 2.1 shows the main dependencies of the properties that were identified. The properties are ordered in a directed graph. The arrow from *situated* to *reactive* for example denotes, that being situated is a prerequisite for an agent to be reactive. The following describes the properties and critically compares the different notions that were found in the literature. An informal definition of the use of the term in this work is given and the relation to other properties is explained.

Autonomy

The property that is probably mostly associated with agents is autonomy. Shoham mentions that the term itself is not precise, but describes it as *[...] the agents' activities do not require constant human guidance or intervention* [123]. Other authors give a stricter interpretation to autonomy. For example Wooldridge describes autonomy in his PhD-Thesis as *[...] the agent operates without direct human (or other) intervention or guidance* [141]. Although both definitions describe the agents independence of the human user the second is probably not acceptable for agents in the sense of expert assistants, since these have to cooperate with the user. It is difficult to think of a productive cooperation without the possibility to intervene.

Jennings shares the definition given by Wooldridge and adds that *it [the system] should have control over its own actions and internal state* [77] — this is the notion that has been adopted by Franklin & Graesser, which base their agent-taxonomy on an *autonomous agent* (see section 2.2.2). That an agent has control over its own actions leads to the question how the agent controls its actions. One possibility is that an agent takes an action or not dependent whether it deems this action to be beneficial to its goal. This would explain why Bradshaw describes the term autonomy as being goal-oriented, although this would lead to the view that goal-oriented behaviour is a prerequisite for autonomy. We will come back to this point later.

Situatedness

Central to all definitions is the ability of an agent to take actions. These actions have an effect on the agents environment, which includes other agents and resources. The relation between the agent and the environment is referred to as situatedness. For Jennings et.al. situatedness means:

> *that the agent receives sensory input from its environment and that it can perform actions which change the environment in some way. [77]*

Situatedness is to be seen as a basic property of agency, since almost all other properties rely on the ability of an agent to sense and act in its environment. Another view of situated agents is given by Nwana in her description of reactive agents. For her agents are situated when:

> *They do not plan ahead or revise any world models, and their actions depend on what happens at the preset moment.[107]*

According to the notion given by Jennings et.al. and the previously cited definitions, we view situatedness of an agent as follows. An agent is situated in an environment if:

1. The agent senses the environment.
2. The agent acts on the environment.
3. The agent actions have an effect on the environment.
4. The agent itself is part of the environment.

The last point leaves room to discussion. Is the agent part of its environment or not? The literature here is again not clear and the different interpretations seem to lead to rather philosophical questions. Franklin states explicitly that the agent is part of its environment. Similar is the definition given by Shoham. In this work we see an agent and its state as being part of the environment, since it can be influenced directly (through communication) or indirectly (through changes in the environment) by other agents, which does affect its behaviour.

Notably Franklin & Graessers definition provides a stricter view of the third point than that mentioned above. For them the agents actions must have an effect on what it senses.

Temporal Continuity

Another basic property of agency is temporal continuity. A concrete definition of this property is again difficult. According to Franklin & Graesser, temporal continuity is the property that does mainly distinguish agents from subroutines and "ordinary" programs. A program usually is invoked to perform a specific task and terminates, when this task is accomplished. On the contrary an agent runs, once instantiated, forever, unless it itself decides not to do [60]. To speak in anthropomorphisms: after an agent is born it lives until it dies naturally or under force. This continuity might be compared with modern operating systems running on a PC that is never turned off, but suspended. Here the system is always available and continuously responding to the user (although it might take some time to wake up).

Bradshaw describes this property as *persistence of identity and state over long periods of time.* As a footnote he mentions that this property ideally includes some sort of episodic

memory. We identified temporal continuity as a prerequisite for being *reactive*, *pro-active* and *learning*. We also want to see persistence of identity and episodic memory as one of the main pre-requisites to enable agents to build up a network of trust. Trust is an important issue, when modelling secure systems.

Reactiveness and Responsiveness

Temporal continuous agents are involved in a constant cycle of sensing their environment and acting. Agents take their actions according to changes in the environment, where a commonly made request is that the action is taken in a timely fashion. The ability of an agent to respond in a timely fashion is described by Jennings as responsiveness:

> Agents should perceive their environment and respond in a timely fashion to changes that occur. [77]

To respond in a timely fashion is the problem that most agents face that utilise traditional AI approaches (i.e. symbolic reasoning). Brooks tried to overcome this problem with purely reactive agents, that do not posses any reasoning abilities. Nowadays hybrid approaches, that combine the benefits of both approaches gain more and more popularity [90].

Communicative and Social

Being communicative is the main requirement for agents to be able to cooperate. For Franklin & Graesser an agent is communicative if it

> *communicates with other agents, perhaps including people. [60]*

Although the communication in purely reactive agent systems is simple and performed through changes in the shared environment, the communication between deliberative agents, i.e. agents that employ symbolic reasoning in their decision making, is more complex and is usually defined at a higher level.

For Jennings et. al. the ability to communicate with other agents, in some form of Agent Communication Language, is an essential part of their definition of agency. Nwana states that the exchange of high-level messages mainly differentiate agent-based systems from other distributed systems.

For Franklin & Graesser social ability and being able to communicate are synonyms. Although a subtile difference can be seen in the fact, that socially able is stronger and does impose that the agents do

> *interact in appropiate situations in order to complete their own problem solving and to help others with their activities. [77]*

This definition differenciates between the capability of an agent to communicate and setting *social* behaviour in corelation with *benevolence*.

Cooperative and Collaborative

According to Nwana [107], the main properties collaborative agents emphasise on are autonomy and cooperation. They work in concert with other agents to achieve tasks on their owners behalf. Typically they are additionally socially able, responsive and proactive. Exhibiting these properties, they are able to operate in open, time-constrained environments.

We distinguish between cooperation and collaboration slightly in a sense that collaboration with other agents is stronger than cooperation. Cooperation can be seen as passively assisting the other agents in their persuit to reach their goals, where collaboration does express that the involved agents share a common goal and are therefore pro-actively taking actions.

The motivation of collaborative agents is that a Multi Agent System of collaborative agents does essemble functionality that is beyond the capability of the individual agent involved in the collaboration.

Strongly related to agent collaboration are the following topics:

- Agent Communication Language
- Agreements
- Multi Agent Planning

Goal-Orientation

Agents are usually serving a special purpose, which means they try to bring about a specific state of the environment – the goal state. Intentional Agents, that do use artificial planning technics, choose their course of action in such a way that they achieve this goal-state. While some researchers take the opinion, that the goal that an agent tries to achieve must be flexible and specified at a high-level [77], others see an agent being goal-oriented when it serves a special purpose to the system [52].

For intentional agents the goal can be specified at runtime, and is specified as a description of the goal-state the agent tries to bring about. For reactive agents the goal is hard-coded in the agents reaction-system. Intentional agents are therefore able to adopt goals dynamically, as a result of collaboration with other agents in the system. In a system of reactive agents the overall behaviour emerges at runtime. The specification of a system-wide goal is therefore a difficult task.

Pro-Activeness

Pro-activeness is one of the properties that is mainly associated with intentional agents. A goal-oriented agent is pro-active, if it takes the initiative and performs actively to reach its goal, by either performing the appropriate actions itself, or by cooperation with other agents.

A prerequisite for an agent to be pro-active is therefore goal-orientation. In the case of intentional agents, the assumption made is that the agents' goals are expressed symbolically and the agent has a model of its environment. If we take view that goal-orientation means that the agent serves a specific purpose, then pro-activeness can be seen in a much more wider sense. Here the minimum requirement for an agent to be pro-active is that the agent is a temporal continuous process, that executes actions in pursuit of its (pre-) defined goals.

This is the view that we take in the development of SANTA agents, where each agent is representing a process, the agent does pro-actively execute actions on observations on its environment (the agents perceptions are reflected in the agents local state). The goal for SANTA agents is defined at a high-level, in the sense, that the actions represent a refinement of possible ways in that the over-all goal (given in the specification) can be achieved.

Flexible

Jennings et.al. [77] see *flexibility* as one of four aspects that define agent-hood. By flexibility they mean that the agent is *responsive*, *pro-active* and *social*. As we analysed before this requires the agent to exhibit a number of other properties, that are not explicitly stated.

The requirement that a program in order to be an agent needs to be *flexible* is therefore strong. Especially, since they take the view that agents are problem-solvers and therefore have a symbolic model of their environment the amount of systems that are Multi Agent System based on this definition are few.

As depicted in Figure 2.1 on page 36, the property *flexible* is a high-level requirement that assumes that a flexible agent exhibits most of the other properties.

Learning

Agents that are learning must be at least *temporal continuous* and *situated*. This becomes obvious when we see learning as the process of gaining experience and derive new knowledge out of experience. To gain experience, the agent must be situated in its environment,

i.e. sense and act on it. By observing the effect that the execution of a particular action has, the agent is then able to employ AI techniques to modify its model of the environment.

> learning means that the agent changes its behaviour based on its previous experience. [60]

Mostly the ability to learn is coupled with a symbolic representation of the agents environment, and often reasoning is based on a formalism that supports the notion of *knowledge* and *belief*. Although there is some well founded theory [116, 123], it is not clear how an agent does actually execute utilising these abstract concepts. But it is conceived, that learning can be achieved without an explicit symbolic representation, for example by using neural networks.

The other major factor that is important to enable an agent to learn is *temporal continuity*, i.e. the agents state must be persistent over some period of time. The emphasis here is that the agent maintains some form of episodical memory.

Collaborating agents offer a wider perspective to learning than the mentioned above. When agents are able to communicate about their past experience and the conclusions that they have drawn from it, an agent is able to learn not only from its own experience, but also from the experience of others. This is mainly investigated in intelligent, especially Belief, Desire, Intention agents, where an agent updates its belief database. For example an agent A sends agent B a message stating that B knows that $X = true$, then agent A could conclude that B wants A to believe that B knows that $X = true$. Any decision that is then based on whether X is true or not, highly depends on the trust, that agent A has in agent B.

This example shows the security implications that come with learning agents. On the other hand, a basic form of learning comes into the picture when we try to address *trust* and *changing policies*. If security policies adapt to changes in the environment, the agents behaviour does so, too. This is a consequence of seeing the set of possible behaviour of an agent restricted by security policies. Trust then inevitably relates back to previous experience.

Adaptive

Franklin and Graesser [60] see *adaptive* as a synonym to *learning*. We mentioned in the previous section some of the implications that come with learning agents. Viewing the complexity that comes when an agent learns from past experience, or other agents then we could view *adaptable* as a further step defining how the agents behaviour changes according to newly learned information.

Bradshaw [33] sees in *Adaptivity* one of the main advantages, that the Agent oriented approach to system development has over more traditional approaches:

> Adaptivity: Agents can use learning algorithms to continually improve their behaviour by noticing recurrent patterns of actions and events. [33]

Relating this again to security would mean that once a network of trust is established, how does it affect the agents security policies. More concretely how can the agent determine whether to grants access to system resources controlled by it to other individuals.

Mobile

Mobile agents are usually cooperative agents, that are capable of migrating through the network and thus change their execution environment. The advantages of mobile agents are mainly on a non-functional side: they can help to reduce communication costs. An often used example is the flight booking agent. In this scenario, the agent does visit several airline-company servers and looks for a cheap flight. Depending on the requirements for the flight-search, the agent is filters the information directly on the server, and helps reducing network traffic. Other advantages of mobile agents are:

- Operating on platforms with limited resources (such as Palmtops)
- Asynchronous computation, while the agent roams the network to accomplish tasks on the users behalf, the user can go offline.
- Dynamically changing configurations, where new agents are made available and offer additional services.

High-Level

We take the attribute of an agent being *high-level* a bit out of the context of the other attributes. This requirement defines that the agent has a symbolic model of its environment and employs symbolic planning techniques. We feel that this requirement, although valid, is vague and not applicable to the wide-spectrum of agents. The main effect of this property is that it limits the different interpretations for the previously discussed properties to those parts that explicitly require the agent to employ a symbolic representation of its environment. Shoham [123] describes *high-level* as follows:

> The high level is manifested in symbolic representation and/or some cognitive-like functions: agents may be "informable", may contain symbolic plans in addition to stimulus-response rules; and may even posses natural language

> capabilities. This sense is **not** assumed uniformly in Artificial Intelligence ... [123]

With this overview of the different properties that agents can exhibit, we will go on and describe different interpretations and classifications of the agents environment.

To summarise: We have shown that, although there is no common definition of what exactly makes a program an agent, several properties are commonly accepted requirements for agent-hood. These are mainly autonomy and temporal continuity. Beside this common ground, one of the key-problems is the different understanding of these properties. Many authors do fear that the lack of a common understanding, what exactly an agent is, poses a threat to the research area as a whole; but a general consensus is, that like in Artificial Intelligence, major advances can be made without a clear and concise definition [52].

2.2.2 Categorisation of Agents

Different attempts have been made to categorise agents. Franklin & Graesser [60] outlined two different approaches to build an agent-taxonomy. Firstly they identify an classification tree that is inspired by biological models (e.g. kingdom, phylum, class, order, family, subfamily, genus, species). Figure 2.2 depicts this taxonomy.

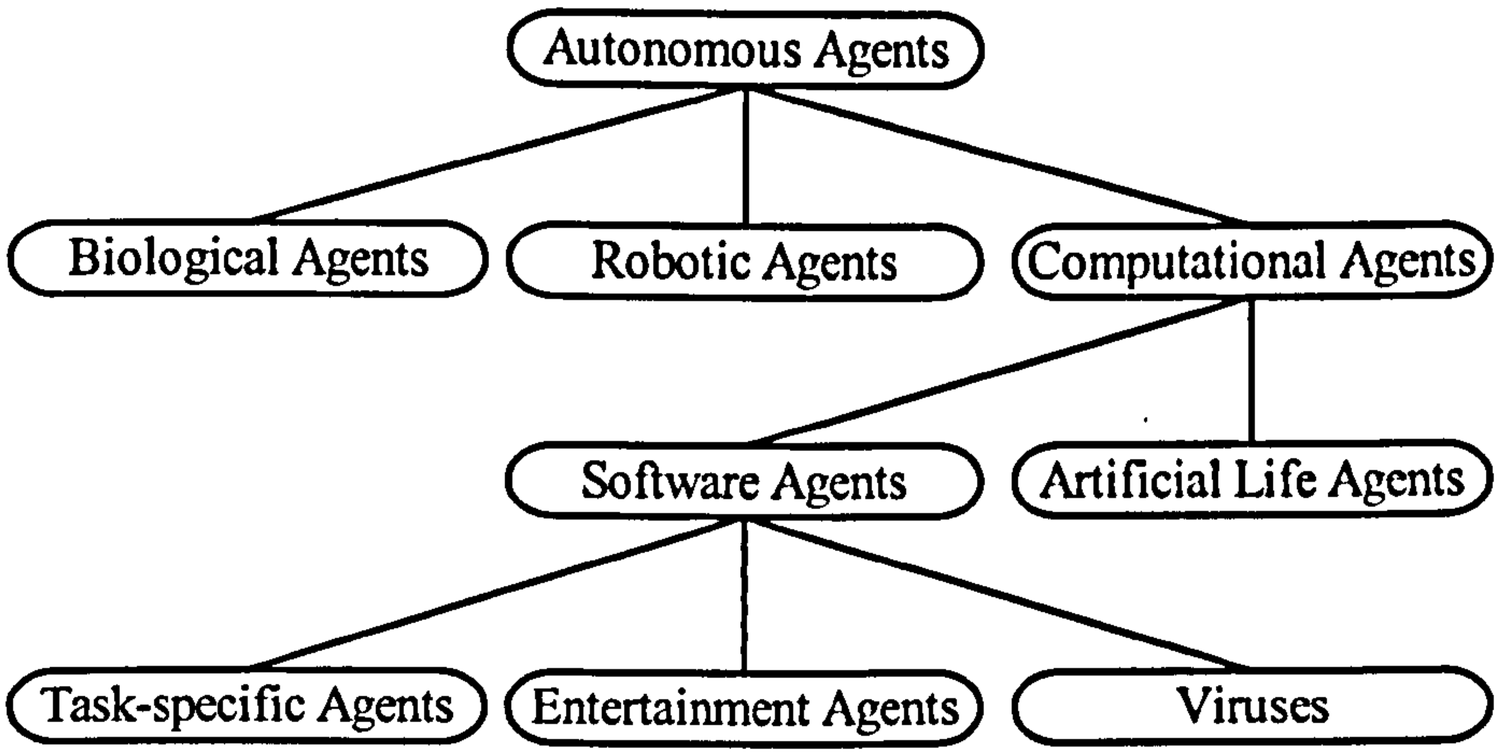


Figure 2.2: Agent-Taxonomy given by Franklin & Graesser

The second approach is to build a n-dimensional matrix of all properties, in which each cell corresponds to a collection of features, and provides one possible category for the classification. As Franklin & Graesser point out for this approach definitions of the properties must be given. The difficulties of this has been discussed in the previous section.

Nwanas shares the view that a typology of agents forms a multi-dimensional space. For a more accessible representation she "collapsed" this space into a list. Obviously this

trades off accuracy against "clarity of understanding". Nwana uses five criteria to classify agents:

1. mobile vs. static
2. deliberative vs. reactive
3. primary attributes (autonomy, learning and cooperation)
4. major roles (e.g. information gathering)
5. hybrid vs. heterogeneous

Interesting is the partial view on the agent-topology that is derived from the primary attributes. Here Nwana describes that for an agent it is necessary, to exhibit at least two of the primary attributes. This is depicted in Figure 2.3.

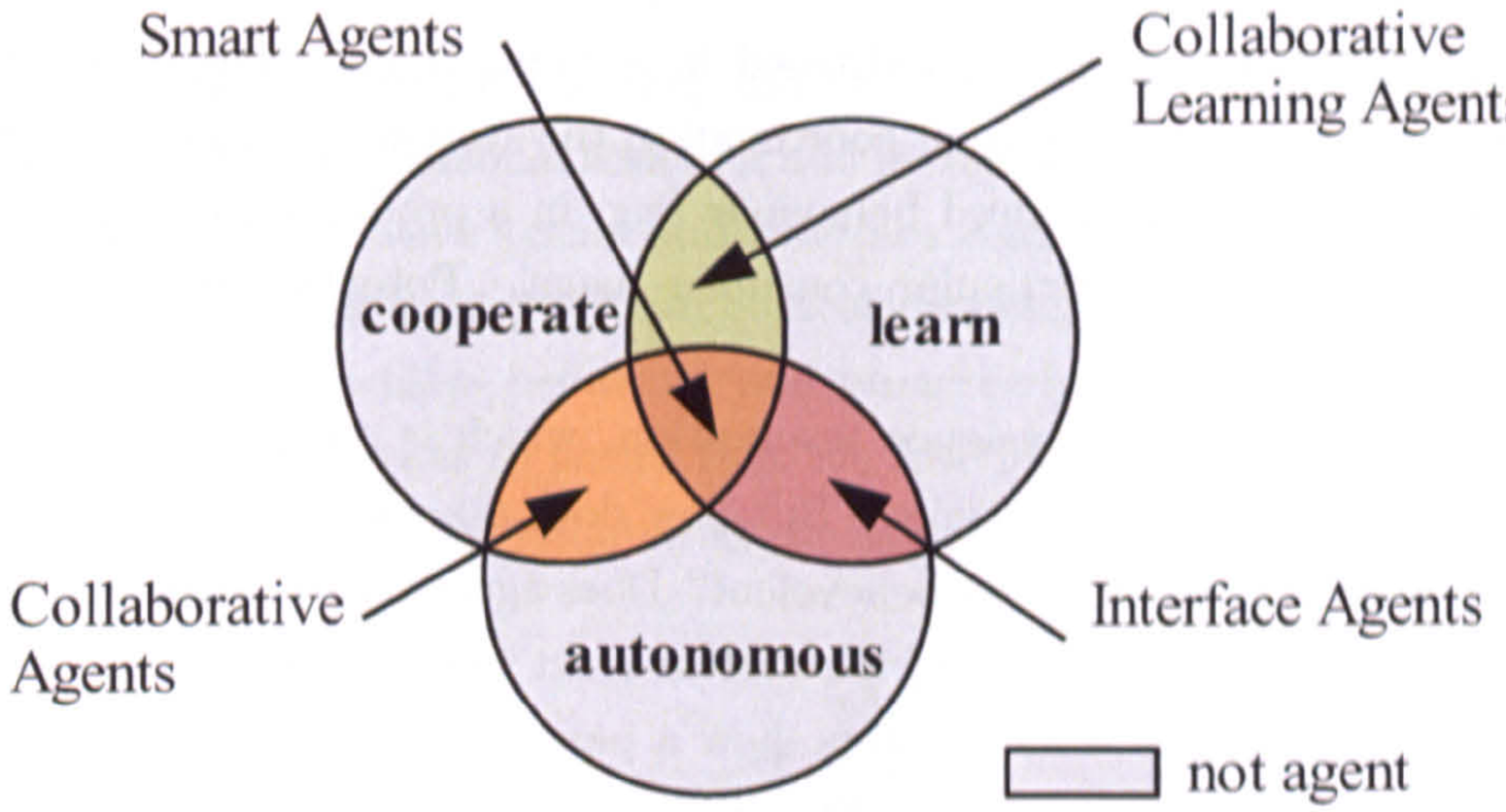


Figure 2.3: A Part View of an Agent Topology (adopted from Nwana)

Derived from the five criteria Nwana presents a list of eight agent types. The classification has been viewed rather controversial and does only claim to *highlight the key contexts in which the word agents is used in the software literature* [107]. The following types of agents have been described by Nwana:

- Collaborative Agents
- Interface Agents
- Mobile Agents
- Information Agents

- Reactive Agents
- Hybrid Agents
- Heterogeneous Agent Systems
- Smart Agents

In the following we shortly summarise the main aspects of the eight agent types that have been identified by Nwana. The information provided is mainly an extract of her paper [107]. Please notice that, although this categorisation has had a wide influence, it is not universally accepted by the agent-community.

Collaborative Agents

Relating back to the analysis of agent properties we have claimed that one of the main aspects for an agent to be collaborative is be cooperative and social. Coordination is one of the key points that need to be addressed in a Multi Agent System of collaborative agents. Without clearly defined agent coordination the system is at risk to enter deadlock situations, starvation or unbalanced behaviour (e.g. in a producer-consumer example all resources are given to one particular consumer agent. Potential other consumers are starved).

Another aspect is the benevolence assumption, which is often made by deliberative agents that employ symbolic reasoning. But how does a network of collaborative agents behave if one malicious agents is not benevolent? Does agent coordination allow to identify self-interested agents, and how should the system react? Game theory and work on agent negotiation tries to provide some answers, how a pareto optimal solution, viz. a solution for which no agent can improve its benefits without decreasing the benefit of another, can be guaranteed by a selecting a specific strategy. One of the commonly known examples is the *Prisoners Dillema*. A brief introduction to this topic is given in e.g. Wooldridge [139] in Chapters 6 and 7.

Wooldridge also addresses in his book agent coordination by *partial global planning*, by *commitments and conventions* and *norms and social laws*. Bergenti et.al. [25] discusses three different approaches to agent coordination: *i*) tuple centres, *ii*) interaction protocols (like contract-net) and *iii*) semantics of Agent Communication Language and highlights their potential benefits and commonly made critics. Omicini et.al. discuss the notion of an Agent Coordination Context in [109] as a *model for the agent environment, by describing the environment, where the agent can interact* and *enable and rule the interactions between agent and the environment, by defining the space of admissible agent interactions*. They later extended their work and relate coordination to security [110].

The existence of one self-interested agent, that does try to prevent the Multi Agent System of collaborating agents from achieving their common goal is a typical security concern. When identifying the need for dynamically changing security policies in agent system we are interested in a similar scenario. Observing the malicious agents behaviour, we identify the agent as being not trusted and change the security policies of the other agents accordingly. This approach is more targeted at detection and adaption, whilst the game-theory approach targets to find a solution under the assumption that all agents are self-interested.

Furthermore Nwana mentions the problems that are arising out of the complexity of dynamically formed interactions and the difficulty that this causes in the analysis and verification of those systems. This issue is even more complicated, when agents in the collaboration are intelligent/learning agents. Similar points are made in e.g. [136].

Interface-Agents

Mainly the properties autonomy, learning and proactiveness are exhibited by Interface-Agents. The agent acts as a personal assistant and is collaborating with the user. The key-difference between collaborative agents and Interface Agents is that the interaction with a human user does not require Agent Communication Language. An interface agent tries to *observe* and *imitate* the users behaviour and proactively assists the user in commonly occurring tasks. It is important to notice that the Interface-Agent is not a layer between the user and the application, but situated at the same level as the user.

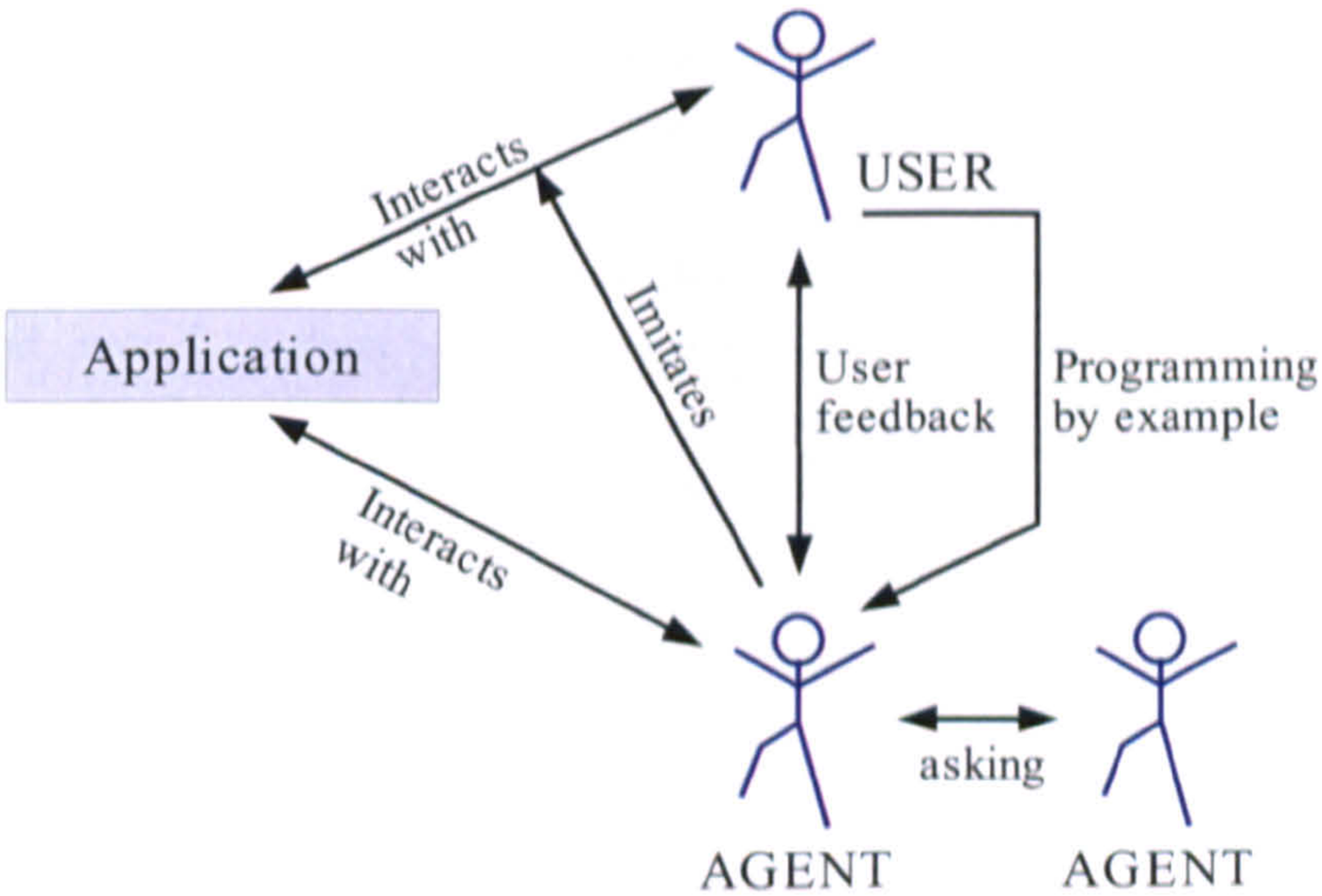


Figure 2.4: Interface Agents (adapted from Maes [91])

The emphasis is here on the learning aspect of the agent, which tries to build a model of the human user and assists in the operation of increasingly complex applications.

- Observing and imitating the user
- receiving user-feedback
- explicit instructions
- asking for advice.

Related to Interface-Agents are the fields of speech-recognition, natural language processing and learning by demonstration. Another important aspect for the acceptance of Interface-Agents is psychology, since the human user needs to understand the actions taken by the assisting agent, in order to build a trust-relationship to their *personal assistant*.

Mobile Agents

One often publicised advantage of the agent approach to software engineering is that autonomous agents can migrate from the originating host machine to other host in the network and then perform tasks there locally. The emphasis on mobility lead to the public perception, that all agents must be mobile agents. When we refer to the analysis of agent properties, this is clearly not the case, rather mobility is to be seen as an independent property that some agent have and others not.

Mobile agents are commonly described as *Mobile agents are processes which can autonomously migrate to new hosts* [54] or as *agents that are capable of suspending execution on one platform and moving to another, where they resume execution* [76].

We depict a network of agents in figure 2.5. In this scenario, the agents are executed on different agent-platforms. A agent-platform defines the local runtime environment of an agent and provides basic infrastructure services, such as Name, Service lookup, inter-platform communication and migration between platforms. In the example agent-platforms reside on different types of hardware: PDAs and a powerful server. Mobile Agents are able to migrate between different platforms and thus utilising the available resources, that are specific to the individual platform.

The benefits and risks of this new approach to distributed computing has been highlighted by Chess et.al. [42]. The main advantages of mobile agents they note are:

- High bandwidth remote interaction.
- Support for disconnected operation.

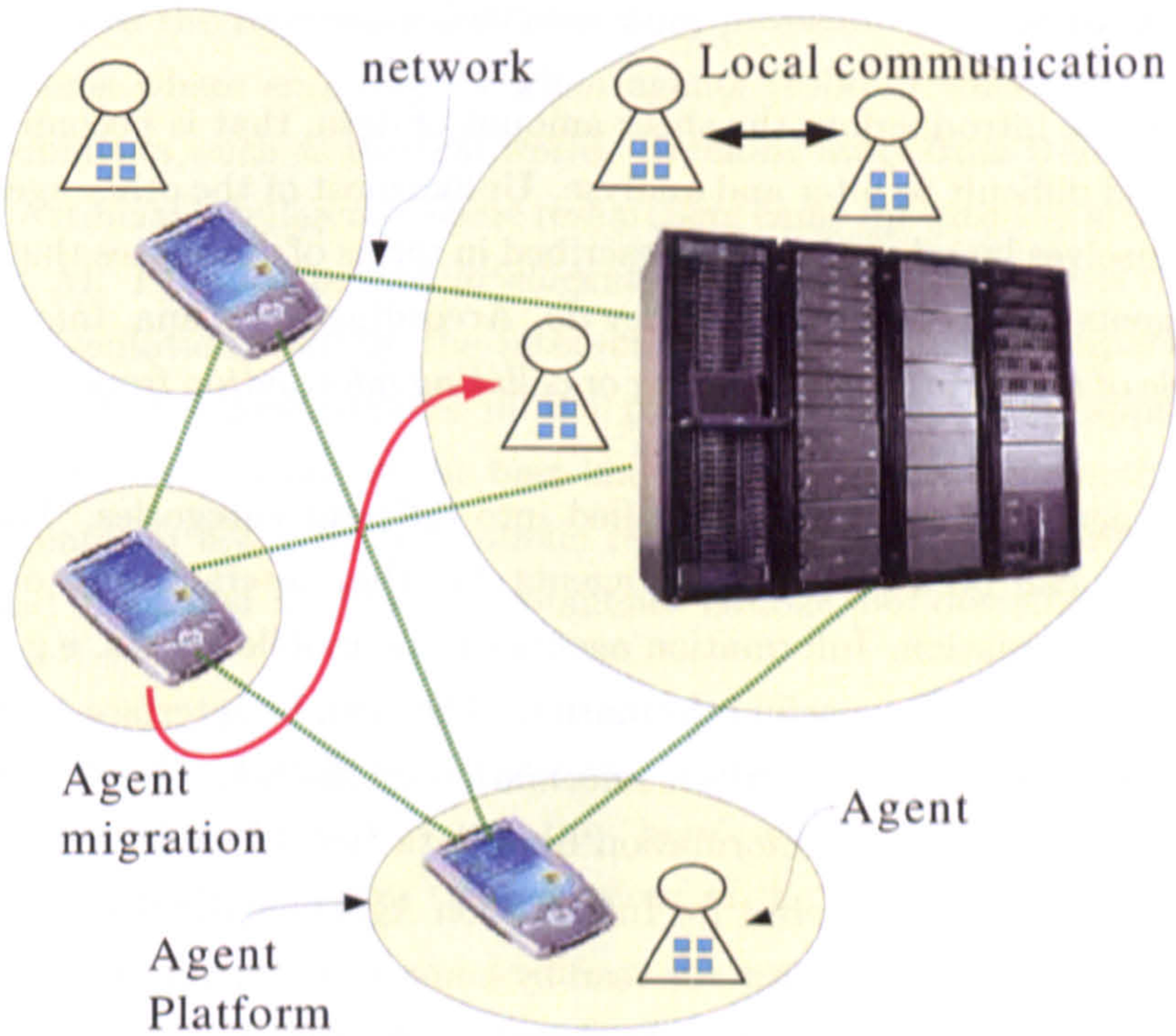


Figure 2.5: Network of agent platforms, mobile agents

- Support for weak clients.
- Ease of distributing individual service clients.
- Scalability.
- Lower overhead for secure transactions.
- Robust remote interaction.

Beside the advantages, the field of mobile agents lead to concerns about the security of such systems. Hosts that accept and execute unknown, possibly malicious code need to protect their resources (host-protection). Mobile Agents need to protect the information they carry against other malicious Agents and also against possibly malicious hosts (agent-protection). The security implications that come with mobile agents are severe and have been addressed plenty-fold.

Most of the Java implementations of agent-toolkits (e.g. Jade/Leap [23, 129, 95], Ajanta [131, 132], ...) allow agents to migrate. Ajanta especially emphasises on the implementation of security mechanisms for mobile agents, where JADE/LEAP focuses on the execution of agents on mobile devices, such as palmtops.

Information Agents

As mentioned in the introduction, the sheer amount of data, that is becoming available is overwhelming and difficult to filter and analyse. Unlike most of the other agent-categories, that define themselves by *what they are*, described in terms of attributes that they exhibit, Information Agents are defined by *what they do*. According to Nwana, *Information agents perform the role of managing, manipulating or collating information from many distributed resources.*

Information agents can often be classified into different categories. For example an Information agent can be a collaborative agents, i.e. they interact with other agents to gather and filter information. Information agents can be mobile agents, e.g. Web-Spiders, that traverse the net to gather useful information. They can be Interface agents, and learn by demonstration, how their owner filters information on the net. They can be intelligent, learning agents, that derive new information from data that is available to them.

An area that is of specific interest for Information Agents is the Semantic Web. While our normal world wide web is meant to be read by humans, the semantic web is marked up, to be read by programs such as Information Agents. The information in the semantic web is structured and a meaning is given to each structure, using the concept of ontologies. Ontologies also play a key role for agent communication in open systems, when agents need to *understand* what other agents mean by sending a specific message.

Reactive Agents

Agents that have a symbolic representation of their environment and that reason about their environment to derive plans for future action have one major disadvantage. Symbolic reasoning techniques do not scale well with increasing complexity of the description. This makes it unattractive for research areas, where real-time decision making is required, i.e. robotics.

Robots must be able to react to certain environmental event in real-time. One example would be the robo-cup event, where robots play football. Imagine an agent that employs rudimentary planning techniques and analyses all possible effects that his action will have on the game. How beneficial planning is depends on the rate in which the agents environment changes. Most planning algorithms assume a static environment, i.e. the environment is only influenced by the agents action. A symbolic reasoning agent in a dynamic environment faces difficulties. Assuming the planning process starts at time t with the environment being in state σ_t and the result of the planning process is available at time t', then its usefulness depends entirely on the dynamicity of the system (i.e. How different is the environment in state σ'_t from the state σ_t). Wooldridge and Jennings in

[140] describe this as the *representation/reasoning* problem.

These problems where encountered when agents (robots) where operating in highly dynamic environments, such as the real world. Breaking away from traditional (symbolic) approaches to Artificial Intelligence, some researchers came up with the idea of behavioural AI, or situated AI. The idea is that *intelligent* behaviour of a system is not founded on a symbolic representation, but by the interaction of the systems components. Rodney Brooks was one of the front-runners in the promotion of this new approach, and his subsumption architecture is one of the best-known architectures for reactive agents.

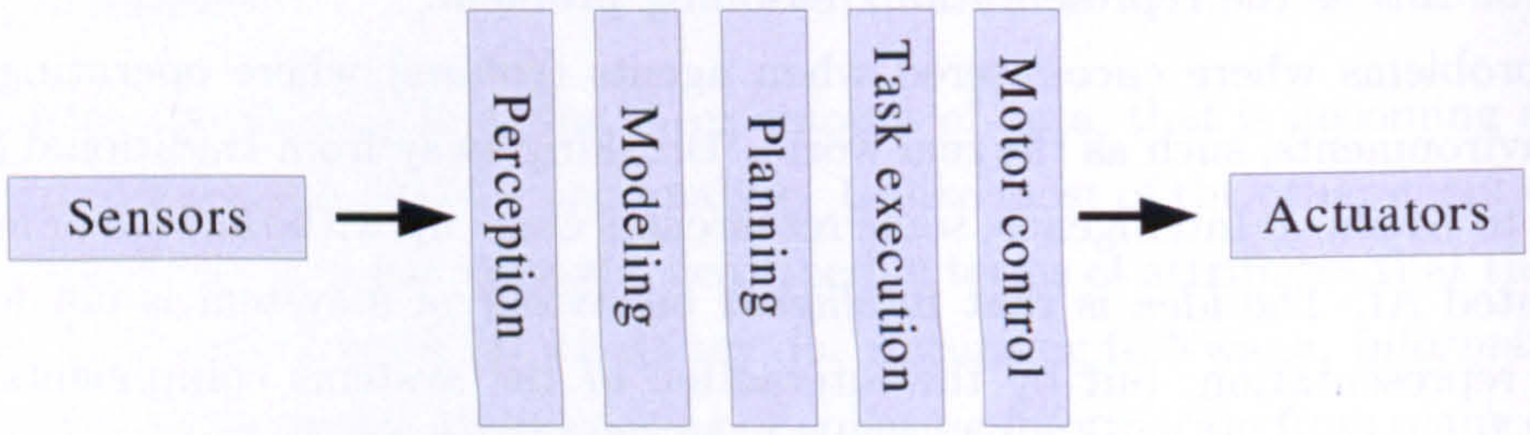
Reactive agents do not have a symbolic representation of their environment. They are situated and embodied in their environment. Brooks [36] describes situatedness and embodiment as:

> **Situatedness** The robots are situated in the world — they do not deal with abstract descriptions, but with the "here" and "now" of the environment which directly influences the behaviour of the system.
>
> **Embodiment** The robots have bodies and experience the world directly — their actions are part of a dynamic with the world, and the action have immediate feedback on the robots' own sensations.

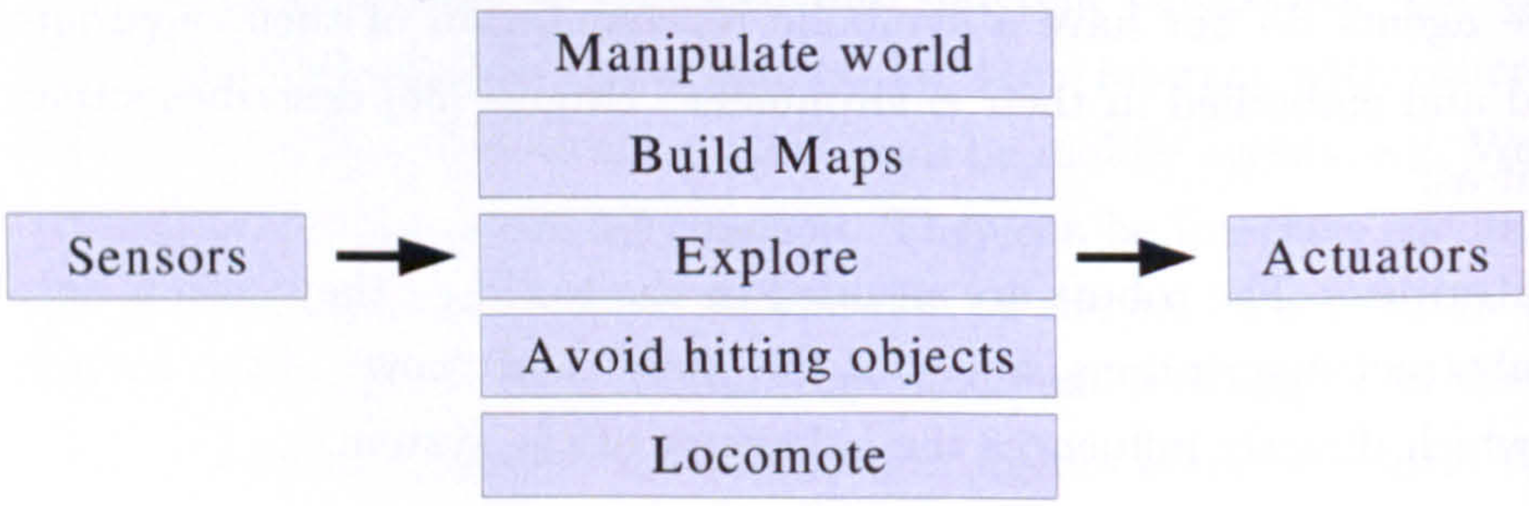
Brooks subsumption architecture decomposes the system *vertically* into different tasks. In the resulting layered architecture each layer has its own level of competence. In general the lowest layer implements the most basic functionality, while higher level control layers represent more complex behavioural patterns. All levels *compete* to control the agent, where higher levels, representing more elaborate goals, can override decisions made by lower level layers. Each level can be seen as a behaviour defined by an augmented final state machine, that is meant to achieve a particular goal. The lowest level in the example below would be "locomote", causing the agent to wander around without any specific goal. This is *subsumed* by the first layer "avoid hitting objects", etc. Figure 2.6(a) depicts the traditional task decomposition used for intelligent control systems, Figure 2.6(b) depicts the behavioural approach.

In the subsumption architecture each layer is unaware of the layers above, but higher levels can utilise data provided by lower layers. Although this approach lead to astonishingly good results, the identification of different layers is ad hoc, and the analysis of the systems behaviour is difficult. Luck states that:

> Systems built on the subsumption architecture are quite complex and idiosyncratic, and it is almost impossible for any formal investigation of the properties of such systems. This makes it difficult to predict or explain such an agent's behaviour in a given environmental scenario. [90]



(a) traditional decomposition of tasks



(b) behavioural decomposition

Figure 2.6: Comparison between both approaches, adopted from Brooks [36]

Most books on agents describe Brooks subsumption architecture as an example of reactive agents. Brooks collection of papers [37] gives an overview of the history of the "new AI".

Although the subsumption architecture defines behaviours at different levels, it does not impose that the agent has any symbolic model of its environment. Hybrid Agents combine both, the deliberative and the reactive approach and thus combine their advantages.

Hybrid Agents

Hybrid agents try to optimise the benefits by combining the different approaches for different agent types. A typical example is the combination of deliberative agents (reasoning agents, that contain a symbolic model of their environment and plan how a specific goal can be achieved) and reactive agents (that are more robust, have a faster response time, and adapt more quickly to changes in the environment). The typical examples that can be found in almost all text-books (e.g. [90] or [139]) are INTERRAP and TouringMachines.

TouringMachines In TouringMachines, the decision process from sensors to actuators is controlled by a three-layered control-subsystem. Each of the three *activity producing*

layers provides suggestions, of what action should be performed. The reactive layer produces a suggestion the fastest and is usually build using situation-action rules, similar to Brooks subsumption architecture. The planning layer is responsible for the agents proactive behaviour, it employs traditional planning techniques and keeps a library of reusable plans for subgoals called *schema*. The modelling layer finally maintains a model of the environment including other agents. It identifies conflicts between agents, and tries to generate new goals (for the planning layer) to resolve or avoid these conflicts. Figure 2.7 depicts the TouringMachines architecture.

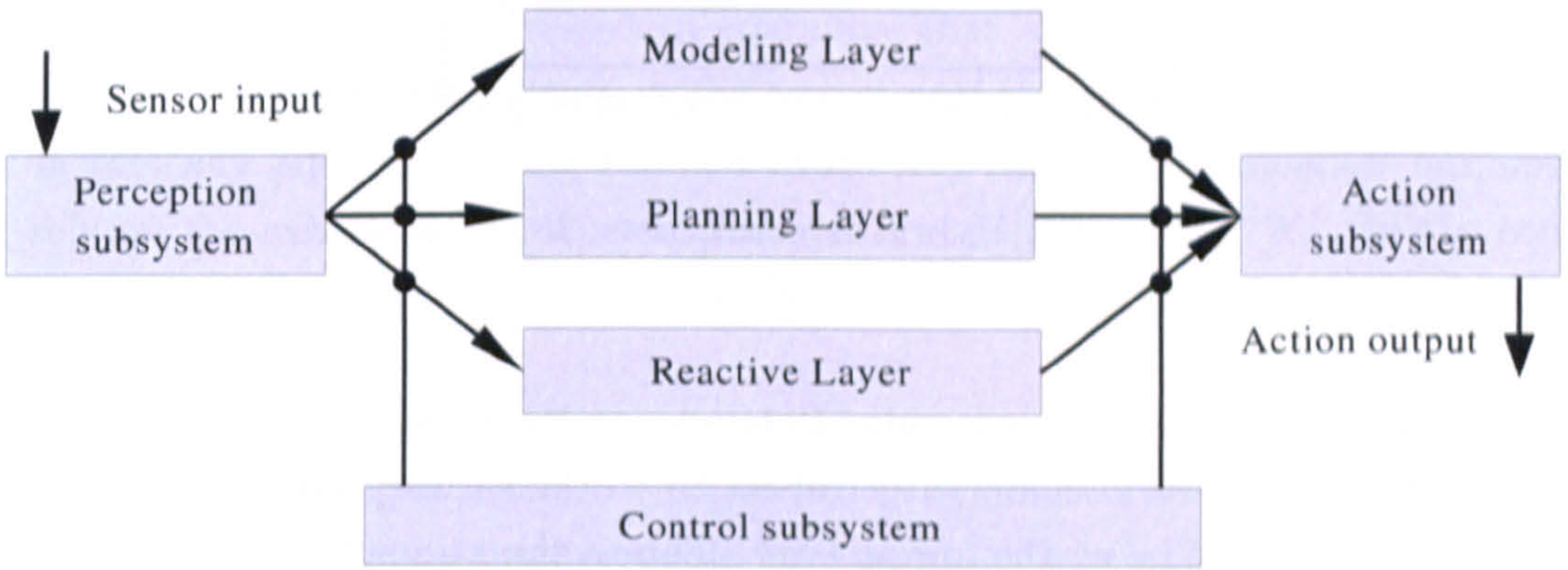


Figure 2.7: Hybrid Architectures: TouringMachines

InteRRaP The INTERRAP architecture provides the similar abstractions as TouringMachines, it is distinct in the introduction of explicit knowledge bases for each of the layers. These represent a model of the environment at the level of abstraction that is appropriate for this level. Another difference is that whilst in TouringMachines conflicting suggestions are resolved by the control-system, in INTERRAP conflicts are resolved by communication between layers. Sensory input is processed at the first layer which is *competent*, and otherwise passed on to a higher layer. This is called *bottom-up activation*. When a layer can handle the situation, it will do so and use *top-down execution* to execute the required action. Thus control flows bottom-up and then top-down the vertical layered architecture of INTERRAP. Figure 2.8 shows the architecture.

Although, hybrid agent architectures combine advantages of both deliberative and reactive approaches for agents, it is still not clear how these systems can be analysed. This is partly because of the reactive layer in the architecture, partly because of the additional interaction between the layer, that make the overall behaviour of the agent more complex.

With respect to the specification and enforcement of security policies, the architectures give material for discussion. At which level are security policies defined? If at a higher

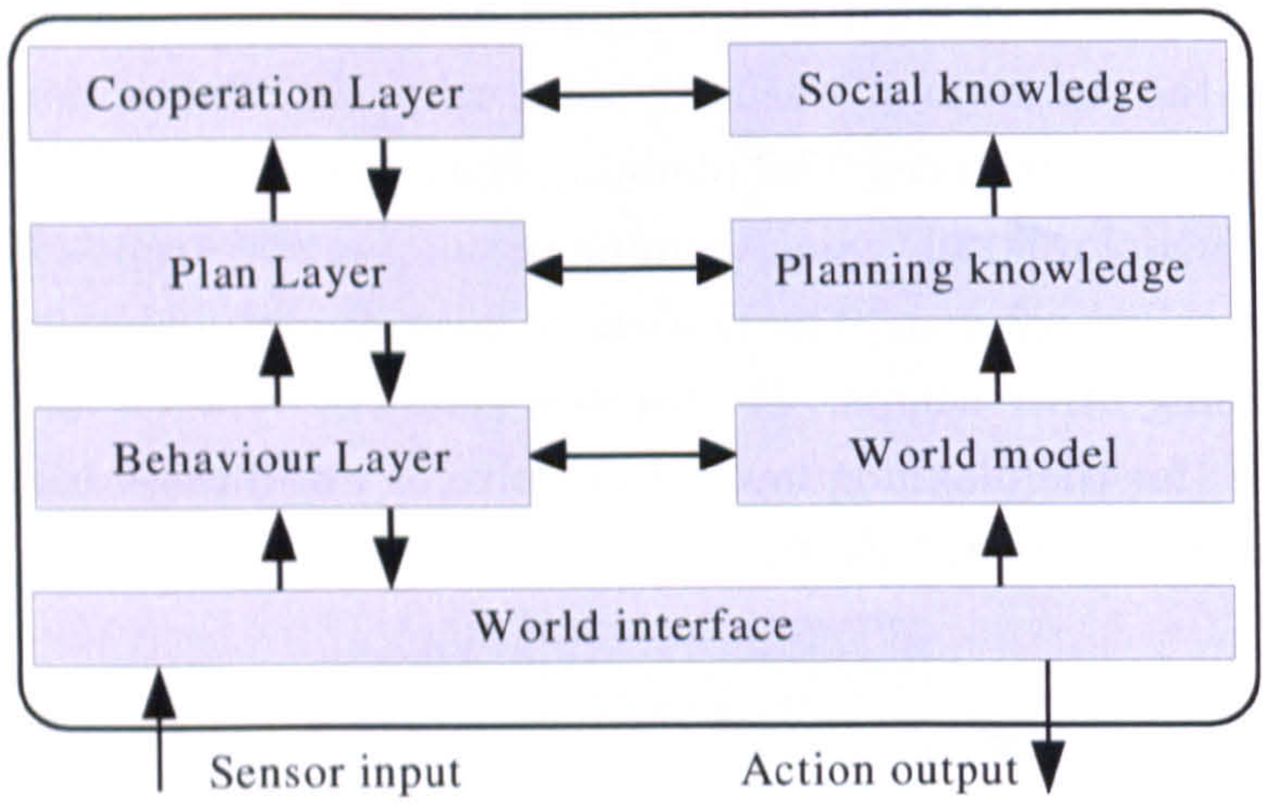


Figure 2.8: Hybrid Architectures: INTERRAP

level, then how are they enforced, since it is unacceptable that the behavioural layer executes actions that are insecure with respect to to the security policy. Consequently the enforcement should be at the lowest level. This is the attempt that we chose using the computational model described in Chapter 3. The enforcement mechanisms of our security policies represent an additional layer that the suggested action must pass before it is performed by the actuators. The specification of the policies themselves is at a higher level (see Chapter 7), requiring to map between the high-level of the policy abstraction and their enforcement (see Chapter 8).

2.2.3 Agents in the Context of this Work

In the context of this work, we are focusing on *reactive* agents and their interaction in a shared environment. We assume that agents are *situated* in an environment, that they can *continuously* observe and affect. Agents react to stimuli from their environment; since we use a temporal logic to describe the execution of agents in our model we can also discuss the timeliness of an agent's reactions and thus address the issue of *responsiveness*. Higher-level properties such as goal-orientation, social behaviour, learning, etc. are not addressed in our framework. However the modularity of our approach means that most agent-architectures can be encoded as part of an agent's *deliberation phase* (see Chapter 3.2.3 for more details).

2.3 Security

In this section we firstly introduce the terminology that is used in relation with the policy and security aspects described in this work. Secondly we survey some of the well known policy models (Section 2.3.1 and discuss critically some of the formal and informal policy languages that have been proposed in this context (Section 2.3.2). Lastly we discuss architectures for the enforcement of policies and formal models that are used to determine the classes of policies that are enforceable.

Security in the context of Multi Agent Systems encompasses a whole set of various problems that need to be addressed to guarantee that a system is secure. Due to the openness of a MAS the first difficulty is the correct *authentication* of agents in the system. This is typically addressed using Public-Private Key-based authentication mechanisms that rely on the existence of a trusted third party e.g. [105, 83, 53, 26]. In the context of this work we assume that all entities in the system can be authenticated and uniquely identified.

Another important issue is the protection of the *integrity* and *confidentiality* of messages, that are exchanged between entities in the system. This is typically addressed using encryption techniques by the middle-ware on top of which the MAS is implemented [80, 133, 87, 22]. In this work we also assume, that the confidentiality and integrity of messages that are exchanged between entities in the system is given and focus on the specification and enforcement of *authorisation*, *obligation*, *delegation* and *integrity* constraints on interactions.

Authorisation Authorisation policies define the conditions under which an authenticated subject (a user, or agent acting on behalf of a user) is allowed to execute a specific action on an object (also called target).

Obligation Obligation policies define the conditions or events (sometimes also called triggers) under which a subject has to perform a specific action on an object.

Delegation Delegation policies define the conditions under which a subject can delegate a specific access right (i.e. the right to perform an action on an object) to another subject.

Integrity Integrity policies define constraints on the execution of an action on an object by a specific subject.

Whilst a system in general provides functionality to a subject, security is concerned with the restriction of the functionality based on the subject's *identity*. A subject is an entity that participates in the execution of the system, e.g. a user, an agent running on

behalf of a user, groups, compounds of subjects or roles. A *group* is a set of subjects, the access-control decision is dependent on the membership of the subject in the group. A *compound* of subjects expresses that two or more subjects act jointly. Examples are procedures to open a bank safe, or activating nuclear weapons, where N out of M subjects must perform an activity jointly, e.g. turning a key, entering a code, etc.

When is a System Secure? We use an axiomatic definition of security. This means a system is defined to be secure if it does not violate its security policy. The security policy defines constraints on the system state (and transitions between two states). Under the *enforcement* of security policies we understand the implementation of *security mechanisms* that guarantee that the system will never be in a state that is not secure.

The enforcement mechanisms can be seen as the link between *security policy*, viz. the specification of security requirements, and the functional requirements of the system. A concrete framework to formally express security requirements is called a *security model*. A security model allows to express certain types of security policies and – if based on a formal language – allows to reason about the policy. Whilst models still describe an abstraction of the system, *security mechanisms* describe *how* the security policy is enforced, i.e how the access to resources is controlled. Security mechanisms manifest themselves in structural design decision and algorithms that determine if a concrete request is processed by the system.

2.3.1 Access Control Models

In this section we overview different approaches to the specification of security policies, where the main focus is on access-control. Anderson [8] describes access control as follows:

> *The function of access-control is to control which principals [subjects] have access to which resources [objects] in the system — which files they can read, which programs they can execute, how they share data with other principals, and so on. [8]*

Based on this definition we view access control as the mechanisms that enforce authorisation and delegation policies. Access Control Models can be broadly divided into *Mandatory Access Control* and *Discretionary Access Control.*

Mandatory Access Control (MAC) In mandatory access control the system's security policy is under the control of a dedicated administrator. Typically this form of access control model is used for *military policies* (also called *governmental policies*). To access a MAC-protected resource the requesting principal must have sufficient

security clearance for the resource's security label. The enforcement of these policies essentially matches the principal's security clearance against the *security label* of the resource. Examples of MAC policies are Bell-LaPadula [86, 21] or the Chinese Wall Policy [35].

Discretionary Access Control (DAC) Whilst for MAC the control of the system's security policy is under the control of the administrator with DAC the control is under the discretion of individuals, e.g. the owner of the resource. DAC is typically present in the protection of data in a (UNIX) file-system, where the owner of the file can change the policy controlling the access to it. This makes DAC unsuitable for military policies, where the confidentiality of information should not be at the user's discretion. To increase the security and prevent (accidental) information leakage most UNIX systems now provide support for MAC (e.g. SELinux [10]).

Non-Discretionary Access Control (NDAC) In [16], Bandara mentions NDAC as situations in which *"authority is vested in some users, but there are explicit controls on delegation and propagation of authority."* In this context *administrative policies*, that control who is authorised to modify access rights are of importance.

In the following we will summarise some of the well known models for access control and emphasise aspects that influenced the policy model that is used in this work. The discussion is based on the original publications, surveys [85, 64, 119, 48, 16], books [31, 8, 24] and comparison papers [130, 4, 102].

Access Control Matrix

A good way of representing access-control specifications is in the form of an access-control matrix. This generalisation of access-control was pioneered by Lampson [84] and later extended by Harrison, Ruzzo and Ullman (HRU model) (introductions are provided in [31, 24]). The HRU extension addresses the issue of subject/object creation and deletion, as well as the dynamic addition and removal of rights. The matrix defines which subjects are allowed to perform which actions on which objects in the system (at a particular point in time).

A reference monitor for example can then directly base the access-control decision on the corresponding entry in the matrix, whilst other enforcement mechanisms need to satisfy this specification by other means. Often the access-control matrix is distributed over the system, to avoid bottle-necks and single points of failure. The most common form is to distribute the matrix to each protected object. Figure 2.9 shows the organisation of the access-control matrix and the distribution to objects. The latter is commonly known as access-control list.

principals	resource			
		R1	R2	...
	P1	{a1, a2, a3, ... }	{a1, a3, ... }	...
	P2	{}	{}	...
	P3	{a4}	{a1}	...
	...	...	...	...

(a) Access-Control Matrix

principals	resource	
		R1
	P1	{a1, a2, a3, ... }
	P2	{}
	P3	{a4}
	...	...

(b) Access-Control List

Figure 2.9: Access-Control Matrix and Access-Control List

Referring to Figure 2.9 an access-control matrix stores the *actions*, that a *principal* (subject) can perform on a *resource* (object). For example in UNIX principals are the users, actions are *read*, *write*, *execute* and objects are files in the system.

The access-control matrix represents a central storage of the access-rights. The central storage does unfortunately not scale well. In a system that contains 1000 user and 10000 files that need protection, the access-control matrix would have $30 * 10^6$ entries. This is difficult to manage and control and might constitute a bottleneck.

A commonly used technique to distribute access-control specifications is in form of access-control lists. An access-control list can be seen as a column of the access-control matrix, that is stored per resource as meta-information. With respect to the UNIX example, the access-control list would be stored on a per file basis. This is particular interesting when access-control is discretionary, i.e. the owner of the files defines the access-rights at his own discretion.

Other options would be to distribute the access-control matrix on a subject basis (rows in the access-control matrix), that describe the *capabilities* of each subject. This is then often enforced by issuing certificates for subjects that use encryption techniques to describe the subjects capabilities. Another alternative to distribute access-control information is to categorise resources in types, and assign principals to groups or roles. In a similar fashion actions could be categorised, which is for example done with access-modifiers (e.g. `private`, `protected`, `public`) in object-oriented programming languages.

Multilevel Security: Bell-LaPadula

The Bell-LaPadula security model was developed by David Bell and Len LaPadula [86] in 1973, and targeted the security of time-sharing mainframe systems. The model is also

known as *Multilevel Security*.

In the model information is tagged with a label, indicating the sensitivity of the information. Traditionally the military classification scheme contains the levels *Unclassified*, *Confidential*, *Secret* and *Top Secret*, but changes to this classification occurred over time and often lead to incompatibilities between different systems. Anderson describes briefly how the classification does change when a unclassified document is transferred from the US to the UK and back again in [8].

Principals hold a specific security *Clearance* corresponding to the classification. An individual is only allowed to access documents that he is cleared for, or documents that require a lower clearance level. The main two principles underlying the model is that:

1. no process may read data at a higher level. This is the *simple security principle* or also known as *no read up*.

2. no process may write data to a lower level. This is the so-called **-property* or *no write down*.

Following the *principle of least privilege* a process can start with an initial low label and increase its security level when it needs to access data at a higher level (*high water mark* principle). According to the *-property it is then impossible to create files with a lower clearance level. This obviously has severe implications on the functioning of the system.

LaPadula mentions an alternative to the *-property. Instead of denying the *write-down* the classification of the file that is written to could be changed. Although this path is followed in the information flow analysis of programs, it is regarded as being counter-intuitive for a security policy model. Some commercial applications exist that follow the approach of *floating labels* Anderson [8] describes them as *products that provide separation more than information sharing*.

Inspired by the Bell-LaPadula model is the Biba model [30] (surveyed by e.g. [85, 119, 24, 31]), that is concerned with the data-integrity of the system. It is often referred to as the *Bell-LaPadula model upside down*, since it exhibits the properties *no write up* and *no read down*. The intuition is that high integrity data cannot be contaminated with low integrity (unreliable) data.

Anderson [8] reviews commonly recognised problems with multilevel security. One of the main problems are *covert channels*. A covert channel is a mechanism that, although not designed for communication, can nonetheless be abused to allow information to be communicated down from a high classification, to a low one. Covert channels are not only a problem in multilevel security, but pose one of its main difficulties.

Beside human issues with the classification and administration of security labels and clearances, Anderson mentions further the necessity of downgrading information in real-applications. Here information must be sanitised, it must be ensured, that it does not contain *hidden* information (e.g. in form of steganography). Another point is that there will always be components, that need to access information at all clearance levels, for example a memory/file management system, or middle-ware in form of Database Management Systems. This can significantly increase the size of the trusted computing base (TCB), increasing the amount of security testing and thus possibly decreasing the trustworthiness of the system.

Multilateral Security and Chinese Wall

Multilateral security, or compartmental security, is not concerned with vertical information flow but with horizontal flow. One approach is to extend the Bell-LaPadula model to a *lattice-model* [51]. The categorisation of the Bell-LaPadula model remains, but additionally all objects belong to a compartment. The classification together with the compartment forms a lattice as depicted in Figure 2.10. The idea is that there should be no information flow between compartments (incomparable nodes in the lattice).

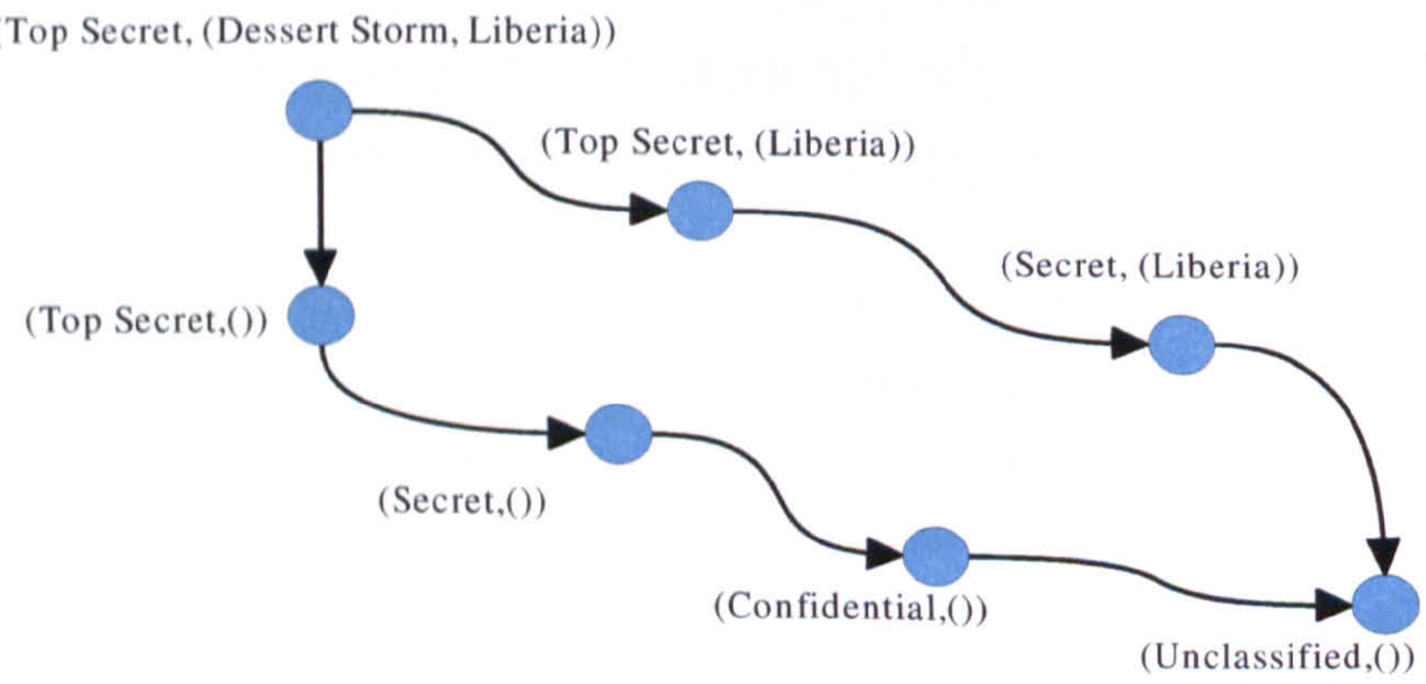


Figure 2.10: Lattice of security labels and compartments

The Chinese Wall Policy introduces dynamic compartments to capture the concept of *separation of duty*. It comes from the financial background, where service firms have internal rules that prevent conflicting interests. The informal example given by [35] explains the Chinese Wall requirement.

> It can be most easily visualised as the code of practice that must be followed by a market analyst working for a financial institution providing corporate business services. Such an analyst must uphold the confidentiality of information

> provided to him by his firm's clients; this means he cannot advise corporations where he has insider knowledge of the plans, status or standing of a competitor. However the analyst is free to advise corporations which are not in competition with each other, and also draw on general market information. [35]

This example cannot be expressed using the Bell-LaPadula model. The key-difference to other models is that the Chinese Wall Policy leaves a subject initially the free choice, which information to access, but restricts further access to compartments, that are *different* to the previous one. This is addressed in the simple security property. The Chinese Wall model is in this way a primitive history-based (or audit-based) access-control model [3] and as such requires more sophisticated enforcement mechanisms than Bell-LaPadula.

The simple security property Access is only granted if the object requested:

1. is in the same company dataset as an object already accessed by that subject, i.e. within the Wall, or
2. belongs to an entirely different conflict of interest class.

***-property** Write access is only permitted if:

1. access is permitted by the simple security property and
2. no object can be read which is in a different company dataset to the one for which write access is requested and contains unsanitized information.

Take-Grant Model

Another fundamental policy model that should be mentioned is the Take-Grant (TG) Model by Lipton and Snyder [89]. In this model the access from a subject to an object is modelled in form of a directed graph. Subjects and objects are represented by nodes. A directed edge from a subject to an object denotes that the subject has access to the object. The access is associated with a weight, that denotes the set of access rights. Fundamental to the model are two access rights [31]:

Take Subject s that has the right to entity e underscores the fact that s can assume any right that e has to other entities such as protected entities.

Grant Subject s that has grant right to entity e can transfer any right it has for other entities to e.

The dynamics of the system is captured by rewriting rules for the graph that can introduce new entities or establish new edges according to the take and grant rights. The analysis

is then based on checking properties of the resulting graphs. [24] provides an example of a safety check for the TG-Model by establishing whether a subject can possibly obtain access to a resource using the graph-transformation rules.

Role-based Access Control

The key to Role-based access control is to abstract away from the individual user and define access-control on the basis of roles in that an individual can act. Whilst group member-ship is static and not bound to change, roles reflect a more dynamic nature. The same individual can be assigned to a specific role temporarily, but whether he is actively acting in the role is a discretionary decision. An example is "a navy officer on night shift".

The rationale for role-based access-control is that roles more naturally express organisation hierarchies and allow the specification of complex security policies with less effort. Whilst the first point is often argued, the second is generally accepted to be valid. This can be shown using a simple example. Imagine, the administrator of the University wide computer network has to define the access-rights for every student separately, instead of assigning just every particular student to the role "student". Of course often groups and group hierarchies can handle situations like this, but roles provide more flexibility as we will show in the following.

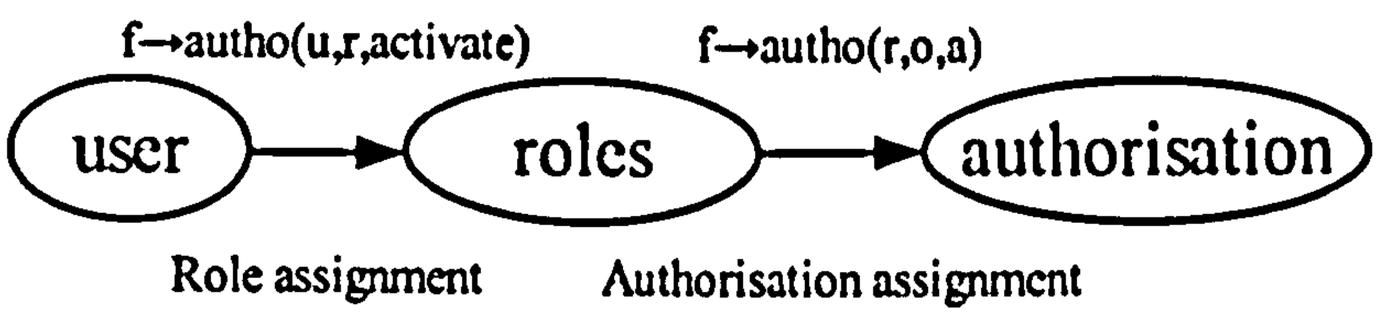


Figure 2.11: Abstraction from user to roles

The abstraction *user - role - permission* is depicted in Figure 2.11. It is one of the main advantages, since it allows to reduce the management effort. For a principal to act in a specific role, two different steps are needed. First the role-assignment, i.e. the principal must be assigned to the role, and second the role-activation, i.e. the principal chooses to act in the role. Where one obvious requirement is that a principal can only activate a role, it is assigned to. The meaning of role-assignment and activation can be constrained, or implicit due to inheritance relations in a role-hierarchy. Both concepts will be explained shortly.

Sandhu et.al. [121] defined four different levels of role-based access-control models, which we briefly summarise below. Starting from a primitive level (0) reaching to the highest, most complex model (3), the four models build the lattice of RBAC model classification.

RBAC$_0$ At this level *users*, *roles*, *permissions* and *sessions* are defined. In conformance with our previous notation, a permission can be seen as a tuple (*object*, *action*). The idea of session is, that a user can act simultaneously in different sessions. The activation of roles and further restrictions, are scoped to the particular session. It is for example possible for a user to activate only the minimal required role for the pursuit of a given task, whilst in an other session requiring a different (possibly higher) role activation. The minimal role activation is often referred to as *principle of least privilege* and can be introduced as a constraint in RBAC$_2$ models. Permissions are assigned to roles, and user are assigned to roles. In this two step assignment lies one of the main advantages of role-based access-control, since it makes the specification of security policies easier.

RBAC$_1$ Includes RBAC$_0$ and introduces the notion of role hierarchies. A role-hierarchy essentially forms a lattice, that specifies the inheritance of roles. A senior role inherits all permissions from the junior role. This further simplifies the specification, but is often a point for critics. For example it is not intuitive, that a manager performs activities (or is even allowed to do so) of an security administrator. The notion of seniority relates more to the object-oriented concepts of generalisation and specialisation and does sometimes fail to capture our intuitive notion, that the role "manager" is senior to the role "security administrator". Figure 2.12 depicts a simple role-hierarchy.

RBAC$_2$ Includes RBAC$_0$ but allows to define constraints on the role-assignment, as well as on the role activation. Constraints define application dependent conditions, and thus allow the security policy to relate to some extend to the state of the system. Separation of duty (c.f. Chinese Wall Policy) can be expressed using constraints.

RBAC$_3$ Finally combines the RBAC$_1$ and RBAC$_2$ models.

Role-based access-control models have been now in the focus of research for some time, but still provide challenges. Extensions to cater for the relationships between roles have lead to the notion of Team-based Access Control [128] and extensions to allow for the specification of temporal constraints on role assignment and activation have been proposed with the Temporal Role-based Access Control by Bertino et.al. [27, 28]. Both RBAC and TRBAC have been modelled by [17] using Constraint Logic Programming. With the goal to express dynamic separations of duty in RBAC, [97] extended the traditional set-oriented specification of RBAC with a version that is based on first order temporal logic. RBAC has also been standardised by National Institute of Standards and Technology (NIST) [9].

RBAC$_3$ models can be expressed using Siewe's [124] basic authorisation framework,

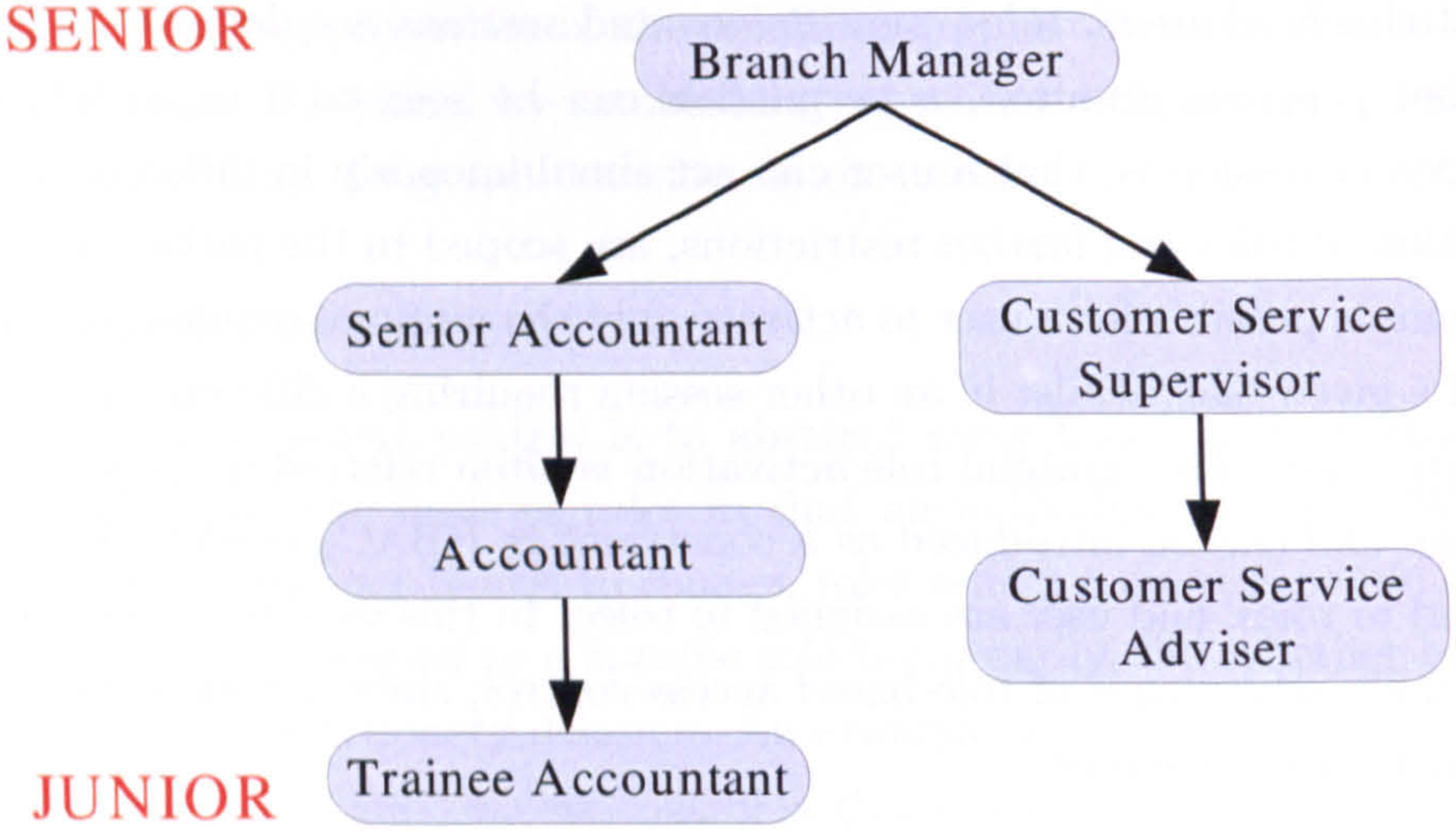


Figure 2.12: Role-Hierarchies

by introducing specialised actions for role-assignment and (de-) activation, together with data-structures that represent the current role-assignments and activations.

2.3.2 Policy Languages

Having reviewed the foundations of security models and policies, we will now review the state of the art in policy specification languages. A policy language is expected to be (according to [20]):

Expressive The expressiveness of a policy language is judged on the informal requirements that can be expressed in the language. Often this does also include the *ease of use*, viz. how complicated are the statements made in the policy language to express a specific requirement.

Clear and Readable The syntax of the language should be readable by humans. [20] argue that for example XML-based languages are too verbose and that most logic based languages are too hard to learn.

Intuitive and Unambiguous Semantics The statements written in the policy language must have a concise meaning. This is typically problematic with languages that do not have a formal foundation, as natural language descriptions will almost certainly contain ambiguities.

Effective Decision Procedure How efficient is the evaluation of the policy to make a decision based on a query.

Extensibility Does the language define extension points to include additional functionality

Except for extensibility these have also been identified by e.g. [67]. Closer to the original foundations we begin with the more theoretical and formal approaches, where the focus is on the unambiguous meaning and then review the more industrial, application-oriented languages that focus on expressiveness and extensibility.

Authorisation Specification Language (ASL)

Let us now review the Authorisation Specification Language developed by Jajodia et.al. [73, 72]. They identified, that most policy languages allow only the specification of a specific type of policy. These are namely:

Open Policies Everybody is allowed access, unless denied. A blacklist is a typical example.

Closed Policies Everybody is denied access, unless explicitly allowed. This is for example the case for Java policy files.

In their work, they proposed the use of positive and negative authorisations rules and show how conflicts are resolved by *decision rules*. They introduce rule-based Flexible Authorisation Manager (FAM) authorisation language. Here positive and negative authorisations for a subject (or group) to perform an action on a specific object can be expressed. The example below shows a positive and a negative authorisation rule:

$$\texttt{cando}(\texttt{file}, s, \pm\texttt{read}) \leftarrow in(s, \texttt{Employee})$$

This rule defines that if subject s is a member of the group `Employee`, then it can `read` the `file`. Negative authorisation would be denoted by a $-$ sign in front of the `read` action. FAM allows also to express authorisations based on previous access using so-called `done` rules. These are essentially facts, that are created by the system (FAM) during runtime and reflect the access executed by a user. This allows for example to express the Chinese Wall Policy. The final decision, whether to grant access or deny a request is then resolved by a so-called *decision rules*. For example the following:

$$do(file, s, +a) \leftarrow dercando(file, s, +a) \& \neg dercando(file, s, -a)$$

This rule specifies that if it can be derived that s is allowed to perform action a on `file`, and it cannot be derived that s is denied to perform action a on file, then s is effectively allowed to perform a on `file`.

Siewe's authorisation model [124] has been inspired by these rules, but he extends the language to allow temporal dependency of authorisations and caters for the compositionality of security policies. One of the major points for criticism is that it is not possible to express obligation policies in ASL. Further it is difficult quantify events, since the done rule merely represents the fact, that such an event has occurred in the past — there is also no notion of time or temporal dependency between the events.

[72] extends the original framework to the Flexible Authorisation Framework (FAF), that includes support for RBAC and discusses propagation rules (for example the propagation of access rights from groups to group-members). They additionally emphasise on integrity rules. However the integrity rules in their model cannot take into account the result of the execution, as integrity constraints are checked prior the decision whether the access is granted or denied - they are mainly concerned with the integrity of the access control policy specification itself. The expression of history-based access control requirements is supported. The history is modelled as a table where each row represents a single access. A row is structured as `(Object, User, Role, Action, Time)`. The history is represented formally by the predicate `done` with a matching list of parameters. It is not clear how the history table is actually updated. The explicit representation of time makes also temporal relations difficult to express.

SecPAL

SecPAL [20] is a very recent authorisation policy language for distributed systems. They claim to balance with their language expressiveness and usability to facilitate widespread adoption. The usability stems from the use of a syntax, that is close to natural language and seems to have been inspired by BAN (Burrows-Abadi-Needham) Logic [38] and subsequent work on access control [2]. An example of a DAC policy in SecPAL is:

Example 1

Admin says user can say$_\infty$ x can access resource if user can access resource.

says here represents an assertion, that is essentially a conjunction of facts. can say$_\infty$ is a delegation of indefinite depth and can is a predicate.

The example represents a policy that allows users to delegate their rights to other users at their own discretion. They claim that languages without recursion (such as XACML) cannot express this simple requirement. The language to express assertions is negation free, to avoid higher computational complexity and undecidability.

SecPAL seems to focus on the delegation of access rights between users, viz. Discretionary Access Control. It is not clear how policies can be composed and how true the

authors claim over simplicity is when large policies are being expressed. In [20] the authors claim that temporal and periodicity constraints can be expressed, however this is not natural to the language. Time is accessible as built-in function **CurrentTime** and constraints are expressed in these terms — the expression of time constraints or temporal dependencies is not a first class citizen.

Becker [20] also notes that most languages that did not start with a formal model as the underlying foundation are prone to semantic ambiguities. The advantageous role that strong logical foundations played in access control has also been emphasised by [1]. However, we would like to emphasise Becker's view, that the logical foundation should not hinder the understandability of the model and the language through its complexity. For example the algebra for policy composition presented in [138], whilst addressing an important subject, appears to introduce a level of complexity that is not justifiable.

After reviewing some of the logic based approaches to the specification of access-control policies, we now have a look at a more informal specification language that is based on a graphical representation of the policy.

LaSCO

The Language for Security Constraints on Objects (LaSCO) [68] expresses authorisation policies as annotated directed graphs. These describe the state of the system and specifies access-constraints. Figure 2.13 shows how the simple security condition of the Bell-LaPadula model is expressed in LaSCO.

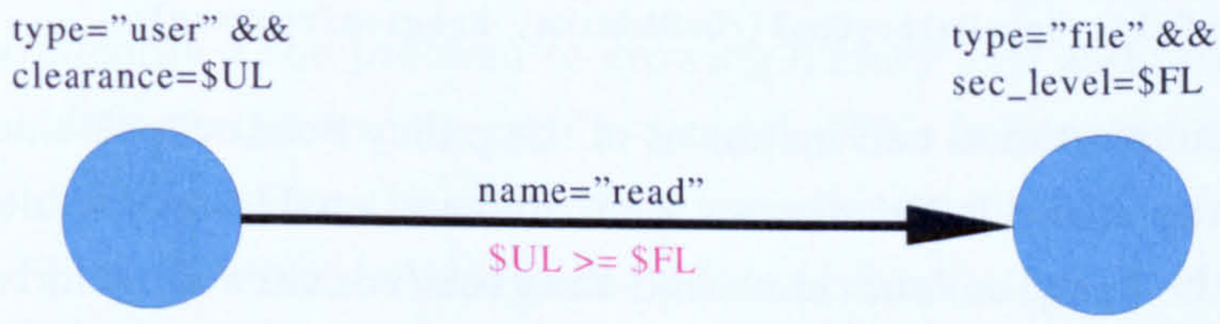


Figure 2.13: Bell-Lapadula's simple security condition in LaSCO

This graph denotes, that an object of type "user" has the clearance level $\$UL$ (a so-called policy-variable) and can perform the action "read" on an object of type "file" with the security-level $\$FL$ under the constraint, that $\$UL \geq \$FL$. An example of the Chinese Wall policy and other more complex examples are given in [68].

Damianou [48] states some of the disadvantages of LaSCO. First it cannot express any form of obligation (which is also the case for ASL), secondly it is not compositional and thirdly there is not textual representation of LaSCO graphs, as it is found for other graph-oriented languages. The main advantage of LaSCO is that a graphical representation

is more accessible to the human user and aids in the specification of security policies. We also believe that the representation of access rights in form of a graph is intuitive for the analysis of permissible information flow in policies. The TG-style notation for access rights features in our Security Policy Analysis Tool analysis tool, that is part of the SANTA workbench.

Ponder

Ponder is a object-oriented, declarative language to express security and management policies. Ponder policies have been implemented using security mechanisms for firewalls, operating systems, databases and Java [47]. Unlike the previously discussed policy models, it supports obligation. In Ponder, objects can be grouped in domains and roles and groups give structure to the systems subjects.

An Authorisation Policy in Ponder defines which subject (group or role) can perform what actions (activities) on a domain of objects. Security Policies are reusable, since they can be parameterised. Thus several instances of the same policy can be created with different parameters. This is one of the strength of Ponder. The example of a simple authorisation policy is taken from Damianou [48]:

Example 2

```
type auth+ PolicyOpsT (subject s, target <PolicyT> t) {
    action load(), remove(), enable(), disable() ; }

inst auth+ switchPolicyOps=PolicyOpsT(/NetworkAdmins, Nregion/switches);
inst auth+ routersPolicyOps=PolicyOpsT(/QoSAdmins, Nregion/routers);
```

The above example shows two instances of the policy `PolicyOpsT` that allow members of `/NetworkAdmins` and `/QoSAdmins` to load, remove, enable or disable objects of type `PolicyT` within the `Nregion/switches` and `Nregion/routers` domain respectively.

Ponder is a flexible and rich language, that allows to define and structure complex security and management policies. The support for roles and groups makes it suitable even for large system policies. Especially the existing tool-support, that allows to graphically view and define the domains and the Ponder compiler, that translates high-level Ponder specifications in low level policy languages, such as the Windows 2000 security templates, or policies for the Java security model, make Ponder an attractive policy specification language.

The three policy languages, ACL, LaSCO and Ponder are used and discussed in the comparative study [4] undertaken by Aljarch and Rossiter. They critically compare the expressiveness of the BMA security model for Electronic Patient Records system developed by Anderson [7, 6]. In their findings they conclude that although, most of the nine

principles can be expressed in LaSCO, Ponder gives the most flexibility to do so. This is partly because Ponder allows the specification of obligation policies.

The Ponder project in the described form is discontinued. However, a successor (re-designed and re-implemented but strongly influenced by the original ideas) is available at http://ponder2.net.

SPL

SPL [118] is a policy language to express access control and a specific form of obligation policies. The main interest of the language is that it addresses policy composition in the sense that many policies can coexists to form the overall policy of the organisation. The composition of multiple policies follows the same lines as discussed in [73, 72]. The advantage of the model is that it discusses the applicability of policies using domain rules, that decide whether a policy applies or not. Policy decisions that are made by SPL can be either *authorise*, *deny*, or *notapply*. Resulting in a tri-valued logic for policy composition. A similar approach, allowing the evaluation of a policy to produce a non-boolean value has also been taken in [18].

SPL provides support for history-based policies [3] and provides a tool that can optimise the history that is stored based on the policy specification. The policy is enforced by a security monitor. The authors show in a separate paper [117] how a restricted form of obligations (requiring the obliged actions to be atomic) can also be enforced using security monitors. This is similar to the approach that is taken in this work. The advantage is that the history that must be kept for the enforcement is kept to a minimum. Other work [61] has also identified the problem of growing history logs and proposed the use of *meta*-policies that define when events can be purged. This is however dangerous, as the deletion of events can mean that the enforcement mechanisms are not able to enforce the policy correctly. Since their model also lacks a formal foundation, it is difficult to show that the meta-policy only deletes events that are not required.

The authors of SPL state as the main drawback of their enforcement approach that *history-based policies cannot decide on events prior to their activation, i.e. the system only records events for each history-based policy after the policy exists.* [118]. A similar problem with respect to the semantics has also been noted by [43]. Our model has the same "drawback", however it is truthful to the semantics of our policy model which explicitly states that the history that is referenced by policy rules is finite, back to the point in time at which the policy started to be enforced.

SPL was not developed based on a formal model and is in this sense comparable with Ponder. It provides many useful constructs for the specification of policies, such as composition, inheritance and instantiations of policies. It also provides the automated

generation of enforcement code. However the lack of a formal semantics of the language makes it unsuitable for security-critical applications.

Usage Control (UCON)

Over the recent years Park and Sandhu developed a model for usage control. The aim of UCON is to provide finer grained control over the usage of digital objects and to unify the access-control, trust-management and digital rights management in one more expressive framework. The initial paper [112] outlines the concepts behind UCON and discusses the different forms of usage in terms of rights combinations. They discuss the accessibility of objects and assert that: ... *since a subject cannot modify an object without viewing it,* ... leads to a reduced number of combinations in their categorisation of accessibility. We feel that this is an overly strong assertion, as it may be the case that a user is allowed to append information to a file without being able to view the files contents. A good example of this is the security requirements for Electronic Patient Records (EPR) [6], where a clinician is allowed to append a new entry to a patient's EPR, *without* reading or modifying the existing entries.

The model is further detailed in [120] where the UCON_{ABC} core models are introduced. The focus is here on the notion of mutable and immutable attributes that are associated with objects or subjects. Mutable attributes can change during the execution of the system, whereas immutable attributes can only change by administrative action. Furthermore the model is extended to capture different phases, before usage, ongoing usage and post usage. The novelty here is that the usage of a resource can be denied whilst the resource is already in use. They also address pre-, ongoing- and post-update functions to comply with obligations and maintain mutable attributes. The definitions that are given in the paper are mostly informal and are not suited for the analysis of UCON policies. Especially reasoning about the interaction of the system behaviour and the policy enforcement seems to be hard, if possible.

This has been rectified to some extend in later work by Zhang [144, 145] who formalised the different UCON_{ABC} models using Lamport's temporal logic of action (TLA) [82]. They mainly focus on the *system-controlled* mutability of attributes, that is *updates are made as side-effects of subjects' actions on objects* [145]. The major advancement of this work is that it provides a formal definition of the UCON_{ABC} core models and their functioning. However it is not clear how policies can actually be *enforced.* The following is the specification of the Chinese Wall Policy in UCON:

Example 3

The first UCON policy states that a subject s is permitted to read from an object o if s did once (at some time in the past) tried to read from the same object and that object

has been at the same point in time in the list of previously accessed objects s.ao.

$$permitaccess(s, o, read) \rightarrow \blacklozenge(tryaccess(s, o, read) \wedge (o \in s.ao))$$
$$permitaccess(s, o, read) \rightarrow \blacklozenge(tryaccess(s, o, read) \wedge (o \notin s.ao) \wedge (o.class \notin s.ac)) \wedge \blacklozenge preupdate(s.ac) \wedge \blacklozenge preupdate(s.ao)$$
$$preupdate(s.ac) : s.ac' = s.ac \cup \{o.class\}$$
$$preupdate(s.ao) : s.ao' = s.ao \cup \{o.class\}$$

The second policy states that if an access request has once been made and the object at that point in time has not been in the list of previously accessed objects and the object has not been in a class that conflicts with s's interests and the actions preupdate have been successfully executed, then the access is granted.

The presented formalisation does only formalise exactly *one* access. It is not clear how the model could support several sequential usages by the same subject on the same resource (object). Semantically this would mean that once an access has been granted in the past it will always be granted in the future. The reason for this is that the ♦ references *some* past state for which the subsequent formula is true[1]. The authors say informally that: "By a UCON policy, we refer to a set of logical formulas for [a] single usage process (s,o,r) all through this paper" [145]. If this means that the policy formalises only one usage (from request to termination), then it would solve the above problem. However it then seems to be impractical (at least) to reason about the effect the policy has on the execution of the system as a whole because the usage processes are independent. In this sense the formalisation using a temporal logic would also not be justified, as the simple state transition diagram (used for illustration in [145]) would capture the sequencing of the usage actions adequately.

Whilst attribute mutability [113] is important and is not often addressed by other access control models it is not clear how one can reason about the interaction between concurrent usage requests, or how these requests are being processed. Especially since *concurrency is a feature in UCON, which has been seldom investigated in access control models* [145]. UCON does also not deal with the composition of policies.

The use of temporal logic specifications to describe access control and other security requirements has been critically reviewed by Calo et.al. [39]. They discuss whether *temporal logic is a good policy specification language.* They do not provide a definite answer,

[1]If all conditions to grant an access have been met in the past, and we want to check at a later point in time whether *permitaccess* can be true then it is still true that once the condition was met.

but emphasise that:

> Policies that are said to be implemented by system *sys* must be expressed in a language from which (at least theoretically) it is computationally possible to implement a monitor that detects when *sys* violates the policy. [39]

They view temporal logic specification as being opposed to basic rule-languages. We feel that Siewe's approach [124] and the approach of encoding rules in temporal logic that can be refined to enforceable code provides a beneficial compromise that inherits advantages from both schools of thought.

eXtended Access Control Mark-up Language (XACML)

XACML [108] is developed as a research project by SUN Microsystems. It is a standard language (XACML 2.0 has been accepted by the Organisation for the Advancement of Structured Information Standards (OASIS)) for expressing access control and privacy policies in the context of a particular XML document. The language itself is based on eXtensible Markup Language (XML). It claims to support role based access control. Bandara [16] argues that the notion of role in XACML is actually the same as a group.

XACML is a rule-based language, where rules define the conditions under which an access to a specific resource is allowed or denied. XACML policies deal with conflict detection and resolution between composed (XACML calls it *nested*) policies, supports the nesting and inclusion of sub-policies by reference. Anderson [5] compares the expressiveness of XACML with the Enterprise Privacy Authorisation Language (EPAL) [69]. She concludes that in terms of expressiveness EPAL is subsumed in almost all aspects by XACML.

The main critique on XACML is that the underlying XML representation is too large and blown up for humans to use directly. The advantage however is that tool support for processing the XML is readily available. XACML comes with an open source reference implementation of a Policy Decision Point (PDP) that can decide whether a specific access control request is authorised or not. Whilst this reference implementation is available, there is no formal semantics for the language itself — which makes policy analysis difficult.

KAoS

KAoS (e.g. [134, 130, 94]) policies have been developed under DARPA and NASA sponsorship, as a collection of componentised agent services that can be used to limit the autonomy in software agent applications. It is compatible with several popular agent frameworks, e.g. Cougaar [44]. Policies are represented using semantic web languages; originally they were represented in DAML [49] and are now expressed in OWL [135]. The original goal

of KAoS is to constrain the behaviour of (semi-) autonomous entities through the use of enforcement mechanisms.

The use of semantic web technology, viz. using the description logic based ontologies as a representation of system, environment and policies allows to exploit the tool-support that is available. For the conflict detection and harmonisation (the resolution of detected conflicts) between policies the Java Theorem Prover (JTP) [59] ontology inference engine has been integrated in the KAoS framework. To make the framework usable and hide the complexity of the policy specification using OWL, the policy development process is supported by the KPAT tool.

Again temporal aspects are only captured in the language using explicit time stamps and are therefore not natural to the model. The dynamic change of policies has been identified as a core issue for the application in MAS. Change here means the dynamic update by an administrator as opposed the dynamic change on events as addressed in our model. It is not clear how one can reason about such changes using the ontological representation. The automated dynamic change of policies has been recognised by the authors of KAoS recently and has been addressed in the KAA framework [34] that extends the KAoS framework with components that automatically change the policies in the MAS, i.e. adjusting their autonomy dynamically. Bandara [16] notes that while the KAoS framework provides support for the conflict-detection and harmonisation of policies, as well as the possibility to perform queries (e.g. "which entities have read/write access to the examination marks database") it cannot take into account the changing state of the system. This is a drawback, especially when the policy depends on the current state of the system. KAoS has been compared by the authors to Rei and Ponder in [130] and more recently in [102].

The motivation to adjust the autonomy of agents in the system originates from the desire to keep the human operator *in the loop*, whilst the system is acting autonomously in a supporting role. The ability to adjust the autonomy provides the human user the assurance that he is in control. The idea of adjustable autonomy has also been addressed in S-Assess [136]. Here the decision-making of an agent is influenced by a constraint-model that is represented as an AND-OR Tree augmented with temporal dependencies between siblings. It is not clear how temporal conflicts are detected or resolved in this framework. However, the main advantage that we see in this work is that the agent's initial decision can be overruled by a policy at run-time. This influenced our model of obligation rules for vigilant agents (see Chapter 7).

We identified the ability to control the behaviour of an (intelligent) autonomous agent in [19] as one of the key issues for the success of agent systems in military applications.

2.3.3 Enforcement

The enforcement of policies means to provide mechanisms in the system, that can ensure that the policy specification is not violated by the system's execution. The standard model for policy enforcement that is used in more industrial approaches to policy enforcement is defined in the ISO standard (ISO/IEC 10181-3:1996) [71]. This model is used for example by XACML and EPAL and has been depicted in Figure 2.14 (adopted from [5]).

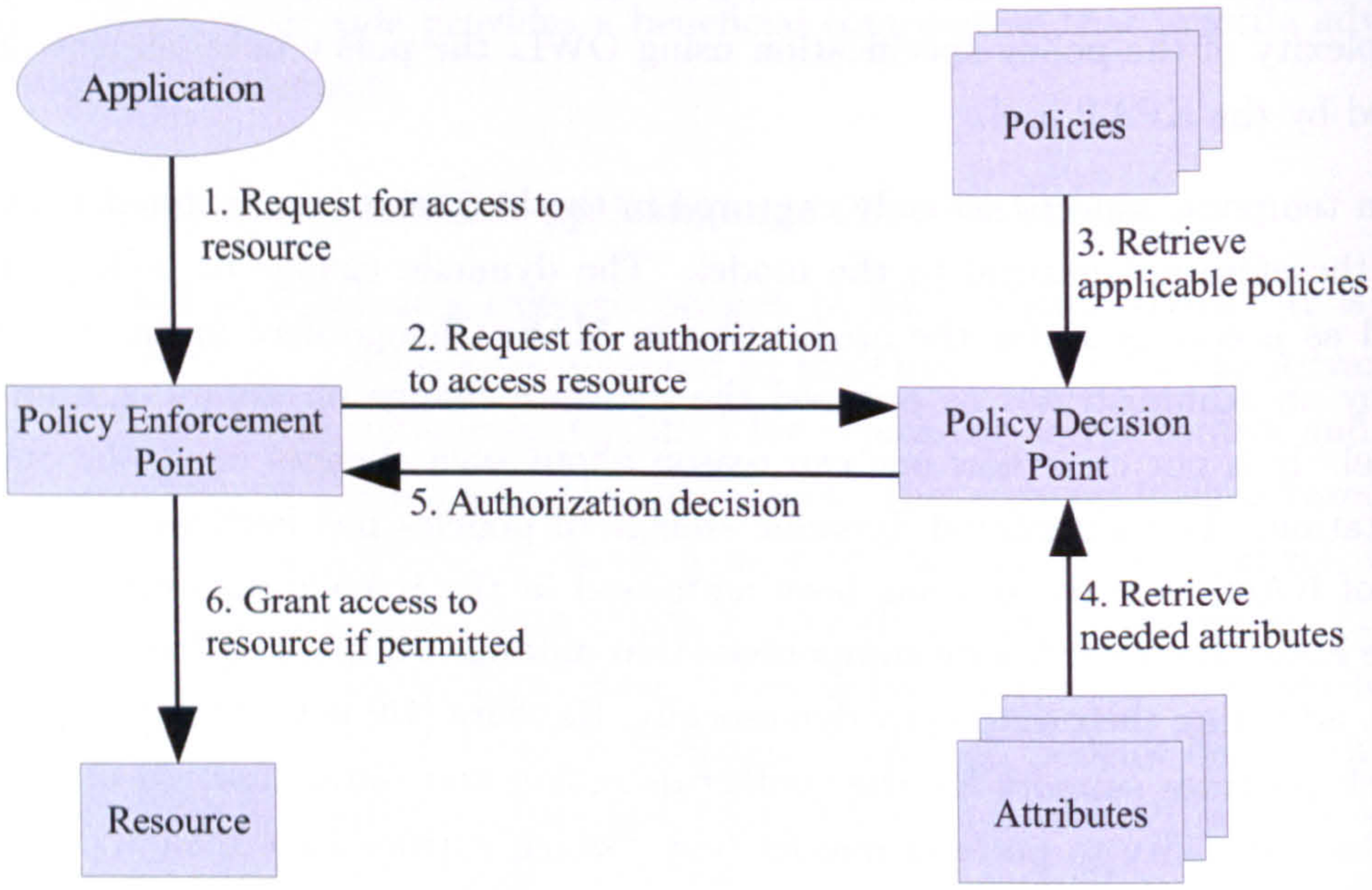


Figure 2.14: Policy Enforcement Model

In this model, an application requests access to a resource. This access is controlled by a Policy Enforcement Point (PEP). The PEP queries a Policy Decision Point (PDP) that interprets the applicable policies and takes the required attributes (e.g. role activations, labels etc.) into account. The authorisation decision is then reported to the PEP that does either grant the requested access or denies it. Obligations in this model are obligations of the PEP to for example make an appropriate entry in an access log file – it does not describe obligations of the accessing application or its user. A similar notion of *obligation* can be found in [126], where obligations denote what a *manager must and must not do.*

The logical separation of PEP and PDP is beneficial as *the implementation of the PEP is usually application- or platform-specific, as it is usually built into the application or the platform on which the application is built* [5]. However, the introduction of PEPs into existing applications has been addressed in for example [111] and [18], where Java classes are edited at the byte-code level during class-loading. The benefit is, that the enforcement code does not have to be manually included, and can even be added to existing programms.

In [18] the performance of this approach has been tested using the Java standard library.

Most policy models and languages focus their discussion on the enforceability on the PDP. They show that the answer to an access control request is decidable and tractable. This is typically achieved by restricting the language/model to constrained Datalog programs (e.g. [29, 72]) that allow to evaluate requests in polynomial time (with respect to data-complexity). Barker et.al. [17] use constraint logic programming to define RBAC and (extended) TRBAC models and for the evaluation of access control decisions. They report results on the efficacy of their approach in [17], however do not provide a comparison to the other approaches.

Schneider [122] discusses the practicality of security languages based on their enforceability and the cost of enforcement using a class of enforcement mechanisms that is called Execution Monitoring (EM). Schneider's approach is based on the notion of security automata (a class of Büchi automata). The security automata reads a sequence of input symbols, that represent system states, atomic actions, higher-level actions (e.g. method calls) etc. Each input symbol triggers a transition in the security automaton. If the automaton does not accept an input sequence it terminates the request. A simple example of such an automaton is given in Figure 2.15 (adapted from [122]).

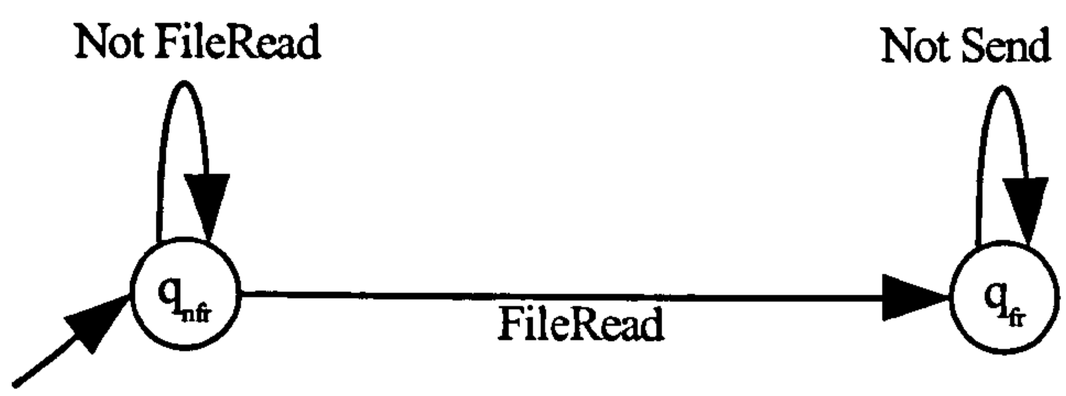


Figure 2.15: No *Send* after *Read*

Schneider notes that the *termination of a target*[2] *that is about to violate a security policy may seem draconian*. However, he argues that the notification of the target about the failure would restrict the set of security policies that can be enforced using EM without apparent gain. This is true from the enforcement point of view, however if we view enforcement as part of an overall software system it is important to notify the target to allow for recovery. Schneider also mentions the possibility of byte-code editing and briefly outlines the concepts of proof-carrying code, that allow to automatically verify that a piece of code satisfies the safety-properties that are stated in a policy [103, 104].

[2] *Target* in the context of the paper denotes the monitored system and *not* the target of an access control request.

The *security automata* approach has been extended by Ligatti et.al. in [88] in form of *edit automata*. They view program monitors as transformers that *edit* the stream of actions produced by an untrusted application. This leads to a hierarchy of enforcement mechanisms. *Truncation automata* can halt the target's execution and are similar to Schneider's *security automata*. Additionally they define *suppression automata* that can suppress unauthorised actions without terminating the target; *insertion automata* that can insert actions into the target's action stream (for example to enforce obligations); and *edit automata* that combine the powers of *suppression* and *insertion automata*. This view of program monitors widens the set of enforceable policies.

2.4 Summary

In this chapter we firstly reviewed the notions of Agents and Multi-Agent Systems, emphasising on the different properties that agents can exhibit. This is not meant to be a universally acceptable classification scheme for Software Agents. The intention is to provide a structured and schematic introduction into the subject. In this work we do not address the higher-level properties of agents, such as learning or collaboration but focus on situated, reactive agents. The agent architecture that is subsequently described in Chapters 3 and 5 is flexible enough to accommodate many of the reactive and hybrid architectures that have been reviewed.

The second part of the review was concerned with security in the context of distributed software applications, emphasising on the concerns that the autonomy of agents introduces. Since the focus of the thesis is the specification and enforcement of access control, obligation and integrity constraints we focussed on policy models and languages for the specification of these requirements. Many languages to define security policies have been proposed. The more formal, logic-based approaches (e.g. ASL) in general lack the flexibility and scalability of more informal specifications (e.g. Ponder, XACML), but have the advantage, that properties of the specification can be proved. The advantage of the informal models is typically the increased scalability, due to concepts like inheritance or instantiation of policy classes as well as the generally better developed tool-support.

We concluded the discussion of related work with the enforcement of policies. The enforcement mechanisms that we emphasised on were those based on a reference monitor model as it can be typically found in operating systems, firewalls and data-bases.

Another important aspect in the field of security, that was not especially reviewed in this chapter, is the impact of more security centric software development processes. The aim to design security into the system is most notably addressed in UML-Sec [79, 78] and the threat analysis in form of (mis-) use-cases [125, 45].

Chapter 3

Computational Model

A Secure Multi Agent System (SMAS) is comprised of Agents, Objects, Policies and Enforcement Mechanisms. This chapter informally describes these components and the interactions between them.

3.1 Introduction

The computational model describes the behaviour of a system and provides the basis to reason about properties of the individual system component's behaviour and their interactions. This chapter introduces the computational model together with the SANTA-WSL design-language. The design-language is used for the specification of a *Secure* Multi-Agent System (SMAS). The aim is to provide a formal and compositional framework in which Multi-Agent Systems can be expressed together with their temporal and security requirements at different levels of abstraction.

Secure in this context means that the system provides adequate mechanisms to enforce dynamically changing security policies that define constraints on the system's behaviour. Constraints are for example access control restrictions or integrity constraints.

A *secure* Multi-Agent System is a distributed system in which the overall system behaviour results from the behaviour of individual agents and their interactions with their shared environment. The system specification is comprised of agents, objects, policies and enforcement mechanisms. The following list provides a brief description of each of these components:

Agents are the actors in the system. Each agent represents a single sequential process. Agents encapsulate a set of variables, that can only be modified by the agent through local computation (*local actions*) or communication with objects (*remote actions*).

Objects constitute the part of the environment that is shared and accessible by all agents. Objects encapsulate a set of variables, that can only be modified through the object's interfaces. Agents *invoke* these interfaces in remote actions to execute the associated code. Objects themselves are passive and cannot initiate any form of computation or communication. Access to objects is mutually exclusive.

Policies constrain the computation and communication in the SMAS. They define the possible behaviour of *Enforcement Mechanisms*. Policies can change dynamically during the system execution, triggered by time and events. Policies are categorised in authorisation, delegation, obligation and integrity policies. They are distributed to the enforcement mechanisms.

Enforcement Mechanisms interpret policies and influence the behaviour of the system by enforcing access control, compliance with obligations and integrity policies. Mechanisms that are encapsulated within another component (viz. within an agent or object) are referred to as *vigilant* mechanisms and those that protect a collection of components are referred to as *security enforcer*.

Figure 3.1 provides a general overview of the SMAS. Agents are depicted as emonicons, objects as rhombi, policies by an earmarked page and enforcement mechanisms as a green shape surrounding the entities under protection.

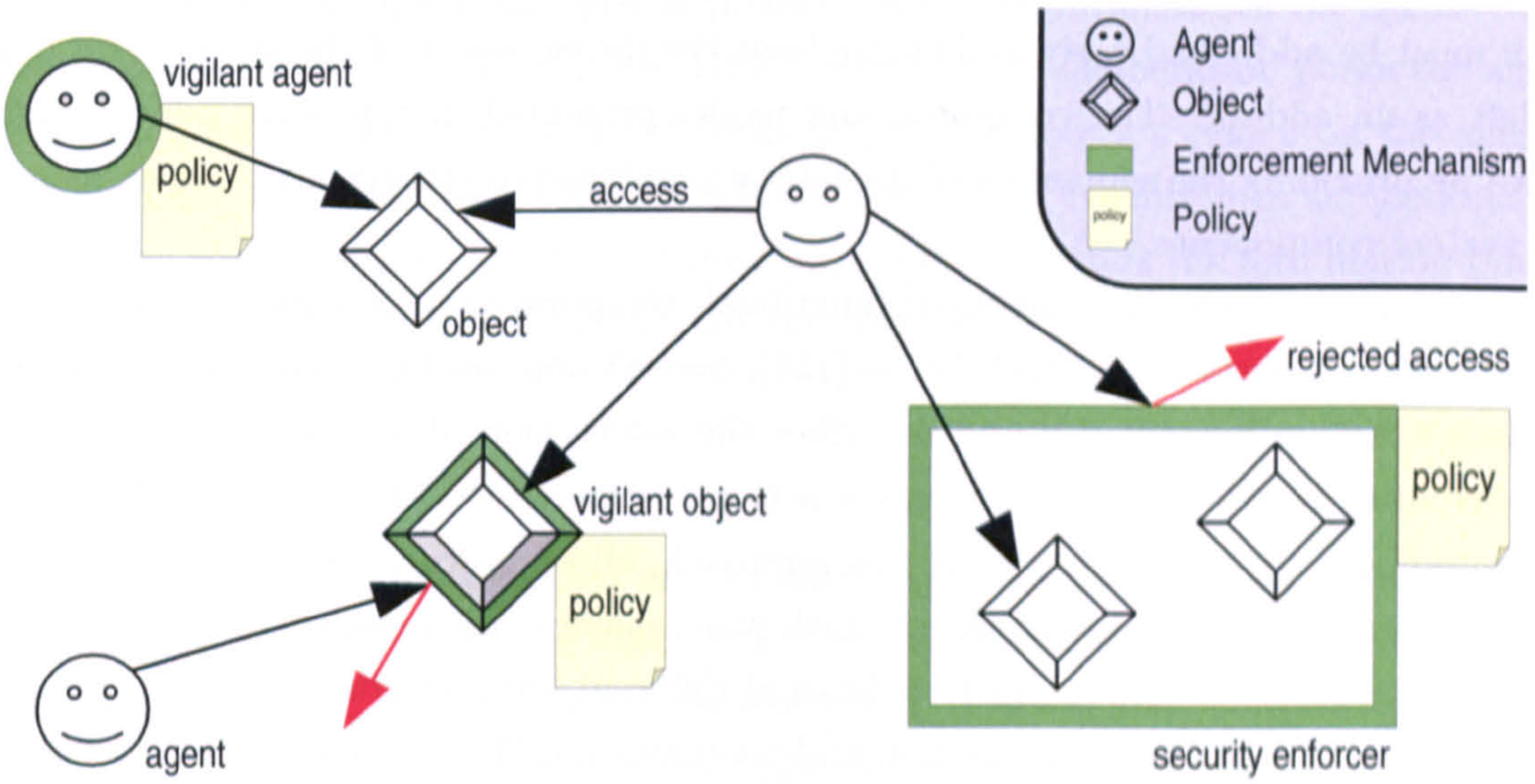


Figure 3.1: General Overview of secure MAS components

Figure 3.1 shows agents invoking interfaces of objects. This is depicted by the arrows. The invocation can succeed or fail. The top-left agent is an example for a successful invocation. The lower-left agent is an example for a failed invocation, indicated by the reflected red arrow. The only forms of failure that are considered in this work are security failures caused by the enforcement mechanisms, e.g. the denial to invoke an object interface. Hard-ware failures and their direct or indirect consequences are not addressed in this work, as we assume that the enforcement mechanisms do not fail. Three different architectures of enforcement mechanisms are depicted in Figure 3.1:

Vigilant Agent Depicted by the green frame surrounding a single agent symbol. The enforcement mechanism is part of the agent's behaviour and controls the agent's internal and external actions.

Vigilant Object Depicted by the green frame surrounding a single object symbol. The enforcement mechanisms are part of the interface definition and are executed when the interface is invoked. The policy is enforced by the object itself.

Security Enforcer Depicted as the green frame surrounding a group of objects. It is a dedicated system process that mediates the interaction between the agent and the invoked interface. It may protect more than one object. The policy is enforced by

the Security Enforcer.[1]

The distribution of enforcement mechanisms in the SMAS and the conception and development of appropriate security policies are an essential part of the system design. Both must be addressed early and throughout the development of the system and cannot be left as an add-on. The computational model presented in this work addresses these issues by providing the semantics of the interaction between enforcement mechanisms and the system components.

Initial work on the integration of functional, temporal and security requirements was presented by F. Siewe in his PhD-Thesis [124]. Siewe's approach is a conservative extension to Back's Action System [12] and describes the interaction of authorisation policies and a distributed system in form of a secure action system (SAS). The computational model presented here significantly extends this approach, allowing for the specification of agent-based systems and the enforcement of much wider classes of policies such as authorisation, delegation, obligation and integrity policies at different levels of abstraction in the system. Additionally the issue of encapsulation and openness of a Multi-Agent System is addressed in this work by defining clear interfaces that control the flow of information between communicating agents.

The remainder of this chapter is structured as follows. Section 3.2 describes the role of agents in the SMAS, the variables that are part of an agent's state, and the execution model of a single agent. Section 3.3 introduces objects as passive entities that represent the agents shared environment. Section 3.4 describes the different types of policies that are used to specify constraints on the behaviour of the SMAS. In Section 3.5 the role of the different enforcement mechanisms in the SMAS is explained.

3.2 Agents

Agents are the actors in the system. They pro-actively act in pursuit of their adopted goals and interact with other agents through their shared environment. Each agent in the system has a unique name by which it can be identified. Authentication mechanisms that verify the identity of an agent are assumed to be in place and the unique name of an agent is used to represent its identity.

An agent encapsulates a set of *agent variables* and a set of actions — the agent's *capabilities*. Agents are autonomous, viz. they are in control of the actions they execute.

[1] A Security Enforcer mechanism cannot protect agents. An autonomous agent is in control of the actions it is executing. If a policy would be enforced on it by a different system component, the execution would be effectively controlled by that component, violating the autonomy property of the agent. Technically the security enforcer can be seen as a proxy object, that is guarding the interfaces of all objects under its protection.

This distinguishes them from passive objects. Consequently, the encapsulated actions can only be executed by the agent itself and not be invoked externally. The encapsulated variables are local to the agent and can only be modified by the agent's actions.

The execution of an agent is staged in phases. After the initialisation the agent enters its Deliberation-Enforcement-Execution (DEE) cycle. In the deliberation phase the agent prioritises actions for execution; the enforcement phase enforces a policy on the agent's decision and the execution phase finally denotes the actual execution of the action. To terminate the agent ends the iteration of the DEE cycle and enters the termination phase, in which it remains idle until the distributed termination of the MAS.

3.2.1 Agent State

Agent variables are the variables that are declared in the agent. They represent the part of the state that is modified by the execution of action statements. *Control variables* are defined at the semantic level to control the execution. They cannot be directly modified by the agent. *Control variables* are also not observable by the agent, with the exception of the boolean variables $done_{a,x}$ and $failed_{a,x}$ that provide feedback to the agent a on the success or failure of the action x and the integer variables $\Pi_{a,x}$ that capture the result of the agent's deliberation phase (see Section 3.2.3). Special constructs to access these variables are provided in the design-language (See Section 5.2.3).

Additionally *auxiliary variables* for the agent deliberation and the vigilant enforcement mechanisms are also part of the agent state. These are local to the deliberation, respectively enforcement phase and cannot be accessed in agent actions. The set of *agent, control* and *auxiliary variables* of an agent are referred to as the *agent's state variables.*

The *agent state* is the mapping from the agent state variables to their values. A behaviour of an agent is a, possibly infinite, sequence of agent states. The set of possible behaviours of an agent is defined by the semantics of the agent in terms of its actions. The semantics guarantees that the agent variables are *only* modified by *one* of the agent's actions at a time. An agent represents a single, sequential process in the system.

3.2.2 Agent Capabilities

The capabilities of an agent are divided into *local* and *remote* actions. All actions are named uniquely within one agent and define a *precondition*[2] and a *statement.* An action can only be executed if its precondition evaluates to *true.* However, the precondition does not imply that the action is executed. The choice of action is also dependent on the

[2]This is a necessary condition for the action to be executed, not a sufficient condition.

deliberation phase and the *enforcement phase*. Only declared agent variables can be used in the precondition and the statement of an action.

Any action can *succeed* or *fail*. The effect of a successful action execution on the agent variables is determined by the semantics of the action. In case of failure the agent variables remain unchanged.

Local Actions

Define the local computation on the agent's variables. The statement of a local action defines the effect of the action, viz. how the agent variables are changed if the execution succeeds. A local action can only fail as a result of the violation of an integrity policy that is enforced by a vigilant enforcement mechanism[3].

Remote Actions

Define the synchronous communication with objects. The execution of a remote action is additionally constrained by the availability of the invoked object. An object may not be available if another agent is currently invoking one of its interfaces, or if the object's security enforcer is mediating the access to another object under its protection. The access to objects is mutually exclusive.

Exactly one object interface is invoked by a remote action and no local computation is performed. The strict separation between computation and communication is made to simplify the computational model and maintain the focus on the integration of security and functional requirements. It is assumed that the order of action executions is determined by the agent's deliberation (and possibly restricted in the enforcement phase). As a consequence transactions must be explicitly encoded in the agent's deliberation. This is not a fundamental restriction to the model and we envision to supply more convenient notations for the specification of transactions in our future work.

Remote actions exchange information between the agent and the object by parameters. The meaning of input and output parameters is defined by an object interface. All parameters are passed *by value*. The effect of a remote action is only partially under the control of the agent. The agent exercises control by defining the input and output parameters of the invocation and can place additional constraints by defining integrity policies. However, the agent does not have control over the actual computation and its effect on the object. A remote action can fail either due to a violation of the agent's integrity policy

[3]All other classes of policies, such as authorisation and obligation, influence the choice of action that is executed, viz. in case of policy violation the action is not executed in the first place and can therefore not fail. Due to the encapsulation and the agent's autonomy, other enforcement mechanisms cannot influence the execution of local actions.

or because of an enforcement mechanism protecting the object. The latter is detailed in Section 3.3.2.

Remote actions define only the communication between agents and objects. Asynchronous inter-agent communication is modelled using these synchronous mechanism.

3.2.3 Single Agent Execution Model

The execution model of a single agent is the continuous iteration of three phases. These are depicted in Figure 3.2.

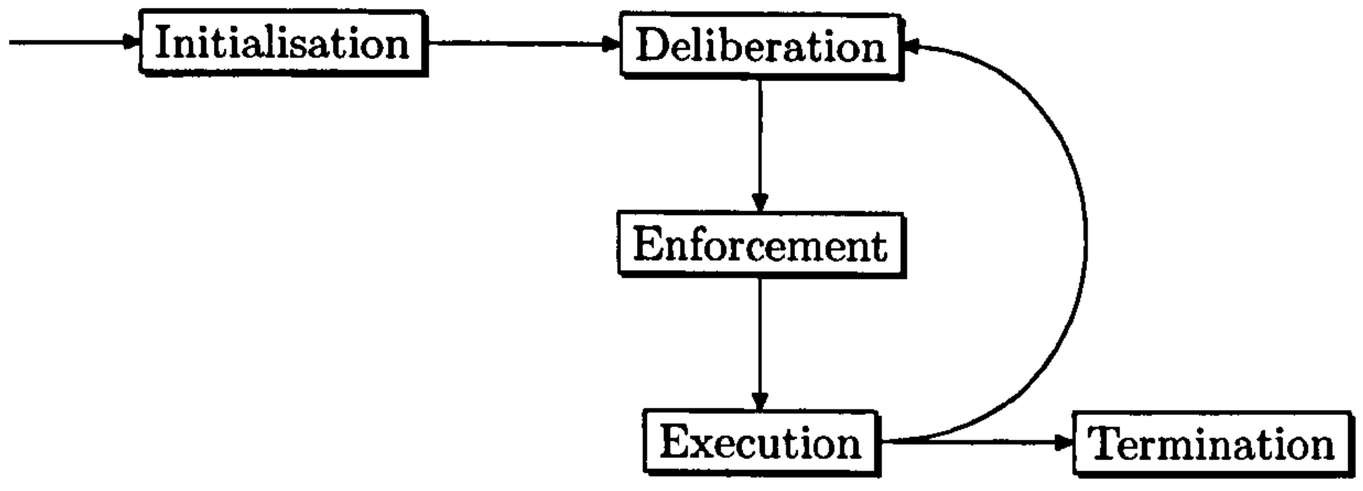


Figure 3.2: Phases in the Single Agent Execution Model

Initially the agent enters the *deliberation phase*. In this phase the agent decides which action it prefers to execute next. It then enters the *enforcement phase*, which maintains the agent's control variables and implements vigilant enforcement mechanisms. Dependent on the outcome of these two phases the agent either enters the *execution phase*, or remains *idle*. It then re-enters the deliberation phase. Alternatively to entering the execution phase or staying idle the agent may also *terminate*, to allow for the distributed termination of the SMAS system.

Deliberation Phase

The *deliberation phase* serves as an abstraction from the typically complex reasoning and decision making process that is found in most agent architectures. To avoid these complexities and keep the model generic, it is assumed that the deliberation phase can be encoded as an *action selection function* that assigns a priority to each of the agent's actions. The intuition is that the action with the highest priority is the one that the agent assessed to be most beneficial to the fulfilment of its current goals.

This approach is more flexible than the selection of *exactly one* action by the action selection function and allows for the anticipation of *failure* caused by vigilant enforcement mechanisms. It also allows for the definition of alternative remote actions for the case that an object is currently unavailable.

Non-determinism between a set of actions can be expressed by assigning the same priority to each of them. During the execution this means that one of them is chosen at random, provided that its execution is not prevented by any vigilant enforcement mechanism in the agent or synchronisation requirements in the case of external actions.

Enforcement Phase

At a high abstraction level the enforcement of security requirements is described in terms of policies and enforcement mechanisms. The enforcement phase in the agent's execution model represents a placeholder for concrete enforcement code that is derived from policies and enforcement mechanisms during the development of the system.

In the *enforcement phase* the agent's vigilant enforcement mechanisms are implemented. They maintain *enforcement auxiliary variables* that are required for the interpretation of policies. A policy is enforced by reassigning the priorities that have been previously assigned in the deliberation phase in such a way that compliance with the policy is ensured. This allows for the enforcement of *authorisation* and *obligation* policies.

Integrity policies are not enforced in this phase, as they typically depend on the result of the action execution. Their enforcement is part of the action execution. The enforcement phase does not modify any *agent variables* or *deliberation auxiliary variables*. The semantics of the enforcement phase does, however, set control variables to indicate that the agent is ready to enter the execution phase.

Execution Phase

In the *execution phase* one of the agent's actions is chosen for execution. The action has the highest priority of all those actions for which the precondition is fulfilled. For *external actions* the choice is additionally constrained by the availability of the invoked object. Actions that meet these constraints are referred to as *enabled actions*.

If more than one action is *enabled*, then the choice is made non-deterministically between these actions. The system ensures the fairness of the action selection and also the mutual exclusion of interface invocations. Fairness here means that if an action is *enabled* infinitely often, it is also executed infinitely often.

The execution ensures that if the action *fails*, the *agent variables* and *deliberation auxiliary variables* remain unchanged. The *control variables* and the *enforcement auxiliary variables* can however be modified by the execution of an action. The former are reflecting that the agent re-enters the deliberation phase and indicate the success or failure of the action; the latter are used to maintain information that is required for the interpretation of policies.

Termination Phase

If the termination of the agent was requested in either the deliberation, enforcement or execution phase using the statement **terminate**, then the agent will leave the deliberation-enforcement-execution cycle and enter the *termination* phase. Termination means that the state of the agent is maintained until the distributed termination of the system. The agent cannot perform any actions and is not accessible.

3.2.4 Agents in SANTA

The SANTA design language provides linguistic support for the specification and implementation of SMAS. The complete syntax together with the informal and formal specification-oriented semantics is described in the subsequent chapters. The listings provided here are not intended to give a full and accurate picture of the constructs, but rather to provide a flavour of how the described components are specified.

Listing 3.1: Agent Definition in SANTA

```
1  agent aid :
2      var1
3      var2 = 0
4
5      when precond1 do action1 : statement
6
7      when precond2 do action2 : object.interface(var1, var2)
8
9      deliberation : external
10 end
```

Listing 3.1 depicts the general structure of an agent. The agent with the name *aid* defines two *agent variables* (var_1 and var_2); a *local action* with the name $action_1$ and an *remote action* with the name $action_2$. The deliberation is in this example not explicitly defined and assumed to be defined externally. This means that the priority values are being assigned any non-negative integer value from 0 to $\texttt{MAX}_{\texttt{INT}}$.

3.3 Objects

The shared *environment* is constituted by a set of *objects*. Objects are passive entities that cannot initiate any computation by themselves. Each object in the system has a unique name by which it can be identified. An object encapsulates a set of variables and a set of parameterised interfaces that agents can invoke. By invoking an interface in a *remote action* an agent exchanges information with the object.

3.3.1 Object State

Object variables are the variables that are declared in the object. They represent the part of the state that is modified by the execution of interfaces. *Control variables* are defined at the semantic level to control the invocation of interfaces. These are not accessible in interface definitions. Additionally *auxiliary variables* for the vigilant enforcement mechanisms are a part of the object state. They are local to the mechanism and cannot be accessed in object interfaces. The set of *object*, *control* and *auxiliary variables* of an object are referred to as the *object's state variables*.

The *object state* is a mapping from the *object's state variables* to their values. A behaviour of the object is a, possibly infinite, sequence of object states. The set of possible object behaviours is defined by its semantics in terms of its *interfaces*. The semantics guarantees that the *object variables* are only modified by the execution of the object's interfaces.

3.3.2 Interfaces

Interfaces define the operations that can be performed on objects. They can be *invoked* by agents and allow for the exchange of information between the invoking agent and the invoked object. The exchanged information is described by the *input* and *output* parameters of the interface. All parameters are passed by value.

The access to the object is *mutually exclusive*, viz. at most one of the interfaces can be executed at any point in time. The invocation of an interface can *fail* due to the enforcement mechanisms that protect the object. In the case of failure all object variables remain unchanged. A failure in the interface execution is also indicated to the invoking agent, resulting in the failure of the corresponding remote action (see Section 3.2.2).

For the analysis of explicit information flow, the invocations are categorised. *Explicit* information flow denotes the information flow that is explicitly defined in the interfaces. It does not address *covered* channels. An example of a covered channel is the fact that the object obtains the information that its interface has been invoked. This is sufficient to transmit a boolean value to the object, without using any of the parameters. We distinguish three different classes of invocations and their combinations for information flow analysis:

Read is an interface invocation that only specifies output parameters. The information flow is is unidirectional from the object to the agent.

Write is an interface invocation that only specifies input parameters. The information flow is unidirectional from the agent to the object.

Query is a *Read* that does not modify any of the object's variables. It is side-effect free.

Policies can be used to control the invocation of actions and therefore restrict the explicit information flow in the system. Given the definition of the interfaces and policies the *permissible* information flow can be analysed. *Permissible* information flow is the flow that is possible under the enforcement of a given policy. The explicit flow of information is always a subset of the permissible flow. This is detailed in Chapter 10.

Policies that depend on the state or history of execution introduce covered channels in the system through the enforcement mechanisms. The introduction of covered channels is mostly not addressed by the literature on state and history dependent policies.

3.3.3 Objects in SANTA

Objects in the SANTA design-language are defined similarly to agents. However, instead of action definitions they define parameterised interfaces. As they are passive, they do not define a deliberation phase, either. Listing 3.2 shows an example of an object definition.

Listing 3.2: Object Definition in SANTA

```
1 object oid :
2     var1 = 0
3
4     interface1 ( in p1, out p2 ) : {
5         statement
6     }
7 end
```

Listing 3.2 depicts the general structure of an object definition. The object with the name *oid* defines one *object variable* (var_1) and one *interface* ($interface_1$) that has one *input* parameter (p_1) and one *output* parameter (p_2). This interface does belong to the classes *Read* (it has output parameter) and *Write* (it has input parameter). It potentially establishes bidirectional information flow. To be able to fully categorise the interface and decide whether it represents a *Query*, knowledge of the *statement* is required to decide whether the object variables are modified by the invocation.

3.4 Policies

Policies constrain the behaviour of agents and objects in the system. They are enforced by the *enforcement mechanisms*. Policies are defined by policy rules and their compositions. Rules typically express a single security requirement and form the basic building blocks of a policy.

Rules consists of a premise and a consequence. The premise describes a set of system behaviours, that when observed lead to the consequence. The consequence of a rule determines whether it is an *authorisation*, *delegation*, *obligation* or *integrity* rule. Rules are composed into policies. The composition operators allow for the dynamic change of policies on time or events and for the scoping and parallel composition of policies. The policy model is detailed in Chapter 7.

3.4.1 Authorisation

Authorisation defines the access to resources in the system and decide whether the execution of an action or the invocation of an interface are permissible. An *authorisation rule* defines the condition under which a *subject* is allowed to perform an *action* on an *object*.

The terms *subject*, *object* and *action* are used in the context of security policies to refer to the different system entities. The term *subject* denotes the actor that initiates a request. In our computational model a subject can only be an agent. The term *object* with respect to policies denotes the target of a request. In our computational model this can be an object or an agent (as local actions can also be controlled). The term action with respect to policies denotes the mode of access to the object. In our computational model actions can be an agent's local and remote actions, or interfaces of objects.

Examples of authorisation requirements are:

- Agent *aid* is allowed to invoke interface $interface_1(p_1, p_2)$ on object *oid* provided that p_1 is less than 10.

- Agent *aid* is denied to execute the action $action_2$ if it previously executed $action_2$.

The first example is a policy rule that protects the access to the interface of an object, whereas the latter is a rule that controls the access of an agent to its capabilities. Both rules require fundamentally different mechanisms for their enforcement. Authorisation rules can define positive authorisations (allowances) and negative authorisations (denials). To resolve conflicts between positive and negative authorisations in the specification, decision rules are used. They determine the outcome of the access control decision.

3.4.2 Delegation

Delegation defines which *subject* can delegate a certain access right to another *subject*. The delegating subject is referred to as the *delegator* and the subject in receipt of the access right is referred to as the *delegatee*. An example of such a delegation rule is:

- Agent *aid* is allowed to delegate the right to invoke interface $interface_1(p_1, p_2)$ on object *oid* to agent aid_1.

Delegation rules are expressed in the model using authorisations on specialised interfaces of a Security Enforcer. Delegation rules restrict the possible delegation of rights, however, the actual delegation must be actively made by the grantor in form of an interface invocation.

Delegation is a powerful concept, which allows subjects to temporarily transfer access rights to other subjects. Delegation policies are important to limit the ability of subjects to do so. Delegation as it is discussed in this work, means only the *delegation of rights*. The *delegation of duties* (obligations) is not addressed and would require further investigation.

3.4.3 Obligation

Obligation defines under which conditions a *subject* has the obligation to perform an *action* on an *object*. For example:

- Agent *aid must* execute the action $action_2$ if it previously executed $action_1$.

This work takes the view, that obligations must be met by the obliged subject itself. This limits the possibilities to *enforce* obligations to vigilant enforcement mechanisms. The enforcement of obligation rules and the implementation of the corresponding enforcement phase of a vigilant agent is discussed in Section 8.5.

Other views on the enforcement of obligations are that the system itself does interfere in the execution and performs the obligation on behalf of the obliged subject. Although this form of enforcement can be expressed in the SMAS computational model, this form of obligation enforcement is closely tied with the *delegation of duties* and not addressed here.

3.4.4 Integrity

Integrity defines constraints on the effect that the execution of actions and interfaces has. They represent assertions on the execution of actions and interfaces. For example:

- The execution of interface *add*(`in` p_1, `in` p_2, `out` p_3) must result in a value for p_3 that is greater than p_1 and p_2.

- The execution of interface *time*(`out` t) must return a time stamp that is greater or equal to the last returned time stamp.

Integrity rules allow for the specification of constraints on execution of an action or interface. If the constraint is violated, the result of that execution is discarded. If all integrity constraints are met, the result of the execution takes effect and is reflected in the agent-, respectively object variables.

The rationale of including integrity policies in the policy framework is that they allow to express expectations on the result of an interface invocation. Unlike assertions these expectations can also be dependent on the history of the execution and thus express a basic notion of *trust*. If the results of a specific invocation did not match the expectations of the invoking agent, then this failure is indicated to the agent that can update its trust levels.

3.4.5 Policies in SANTA

Policies in SANTA are expressed as a separate construct.

Listing 3.3: Policy Definition in SANTA

```
1  policy pid :
2
3      allow(aid, oid, interface_I(p_1,p_2)) when 0 : (p_1 < 10)
4
5      decide(S,O,A) when 0 : allow(S,O,A)
6  end
```

The policy *pid* represents the first authorisation example that was given in Section 3.4.1. It contains a positive authorisation rule, that grants the agent *aid* the right to execute the interface $interface_I(p_1, p_2)$ of object *oid* provided that the parameter p1 is *now* less than 10. The operator ":" is used to define the length of the observed interval; a value of 0 means *now*. The second rule resolves potential conflicts and states that if a positive authorisation can now be derived then the access is granted by the mechanism enforcing this policy.

3.5 Enforcement Mechanisms

Each enforcement mechanism that is deployed in the system can only enforce policies with a certain *scope*. The scope limits the application of the policy and also restricts the access to the observable variables. For example a *vigilant enforcement mechanism* in an agent can only enforce policies on that particular agent. The set of variables that can be used in the policy are also restricted to those that are observable by the enforcement mechanism. Consequently the choice of enforcement mechanisms and the specification of policies must go hand in hand as not all policies are enforceable with all mechanisms. Additionally the specification of behaviours (sequences of states) in the policy is dependent on the chosen enforcement mechanisms as it defines the granularity of time over which the policy is interpreted (see Chapter 8).

3.5.1 Vigilant Agents

Vigilant agents can enforce authorisation, obligation and integrity policies. The scope of *subjects* in the policy is limited to the agent itself. This means the enforced policy can describe constraints on the agent's behaviour only and does not have any direct influence on other agents or objects in the system.

Variables that can be used in policies that are enforced by this mechanism are the *agent variables* and observable control variables and parameters of interface invocations made by this agent. An example of a rule that is enforceable by this mechanism is the obligation example in Section 3.4.3.

3.5.2 Vigilant Objects

Vigilant objects can enforce authorisation, delegation and integrity policies. The scope of objects in the policy is limited to the object itself. This means the enforced policy is limited to the object itself and does not directly influence decisions made for other objects, nor enforces obligations of subjects in the system.

Variables that can be used in policies that are enforced by this mechanism are the *object variables*, observable control variables for this object and parameters of the controlled interfaces. An example for a policy that is enforceable using this mechanism was given in Section 3.4.5.

3.5.3 Security Enforcer

The security enforcer can protect a group of objects and is representing a centralised architecture. It can enforce authorisation, delegation and integrity policies for objects in that group. The scope of the policy is limited to this group. As it is the case for vigilant objects, this mechanism cannot enforce obligations on subjects.

The choice of distributing security enforcer mechanisms in the system will strongly depend on the distribution of the objects themselves. From a specification point of view a security enforcers can enforce policies for objects that are distributed in the system, however implementation consideration will in most cases restrict the scope to a single network node. Other considerations for the choice of distribution are performance considerations. To avoid performance bottlenecks it can be advisable to place independent (from the viewpoint of the policy specification) groups of objects under the control of different security enforcers. This, however, restricts the enforceable policies as the variables that can be referenced in policies are limited to the control variables of objects in this group and input and output parameters of the controlled interfaces. To maintain the encapsulation of objects, access to the *object variables* is not possible.

3.5.4 Enforcement Mechanism in SANTA

Enforcement Mechanisms in SANTA represent the association of a policy with the concrete mechanism for the enforcement. This association allows to check the constraints on the policies to be enforceable by the mechanism statically. The following listing provides three examples of how a policy can be associated with a vigilant agent, vigilant object, or a security enforcer.

Listing 3.4: Examples of Enforcement Mechanisms

```
1  /* Vigilant Object */
2  enforce pid with oid
3  /* Vigilant Agent */
4  enforce pid with aid
5
6  /* Security Enforcer */
7  securityenforcer SE :
8     protect (oid1, oid2 /* , ... */)
9  end
10
11 enforce pid with SE
```

Line 2 states that the policy with the identifier `pid` is enforced by the enforcement mechanism implemented in the vigilant object `oid`. Similarly Line 4 states that the policy `pid` is enforced in the vigilant agent `aid`. Line 7 to 9 define a security enforcer with the name `SE` that controls access to the objects `oid1` and `oid2`. Line 11 then associates the policy identified by `pid` with the security enforcer `SE`.

3.6 Summary

This chapter provided a brief overview of the computational model and the SANTA design-language. It described the components that define the behaviour of the SMAS:

- *Agents* are active and execute sequentially *local* or *remote actions* (computation, respectively synchronous communication). The execution model consists of three main phases: a) Deliberation: Prioritisation of actions; b) Enforcement: Constraining actions for execution; and c) Execution: Execution of the selected action.
- *Objects* are passive and can be accessed through defined interfaces. They are protected by either vigilant or security enforcer mechanisms. The access to objects is mutually exclusive.
- *Policies* define constraints on the execution of the SMAS. They are enforced by enforcement mechanisms. Policies can dynamically change over time and on events.

Policies are expressed in terms of rules that can depend on behaviours that have been observed in the past. They are categorised as: a) Authorisation: who is *allowed* to execute what; b) Delegation: Who is *allowed to delegate* what; c) Obligation: Who *must* execute what. c) Integrity: *Constraints on the effect* of the execution.

- *Enforcement Mechanisms* enforce policies. They are distributed in the system: a) *Vigilant Agent*: the policy is enforced as part of the agent's behaviour; b) *Vigilant Object*: the policy is enforced as part of the object's behaviour; c) *Security Enforcer*: A system process mediating the interaction.

The benefit of the approach is that it integrates the specification of functional, temporal and security requirements in a uniform formal framework. Policies cannot be seen independent of the system, as their interpretation and enforcement is highly dependent on the observable events in the system. This means that reasoning about policies that depend on the history of the system execution and events is only possible if one can also reason about the system itself and the influence that the enforcement of policies has on the behaviour of the system.

The following chapters formalise the computational model and give the specification-oriented semantics of the SANTA design-language. To convey the semantics in a more understandable manner incremental subsets of SMAS are defined that gradually increase the complexity of the model. Chapter 5 formalises the execution model of agents and their local computation. Chapter 6 then extends this by introducing objects and communication. Chapter 7 formalises the SANTA policy model. The formalisation of the enforcement mechanisms is given in Chapter 8, and the refinement of policies into concrete mechanisms is discussed in Chapter 9.

Chapter 4

Preliminaries

In this chapter we provide an introduction to Interval Temporal Logic (ITL) as the underlying logic of our formal framework.

4.1 Introduction

Interval Temporal Logic (ITL) [40] is a flexible notation for both propositional and first order reasoning about periods of time found in descriptions of hardware and software systems. It can handle both sequential and parallel composition unlike most temporal logics. There is a very powerful and practical compositional proof system for ITL [98]. That is, much of the proof of a system specified in ITL can be decomposed into proofs of its parts. It offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and timeliness.

Our choice of ITL as the underlying logic to our formal framework is mainly founded in its compositionality. Especially noteworthy is the sequential composition which is natural to procedural programming languages that we aim to be the target implementation languages of our Multi-Agent System. Further, being a temporal logic it allows us to naturally reason about time. We use this for example to give guarantees on the time the enforcement of a policy does take. The ability to easily map between specifications defined at different granularities of time is a great advantage in the specification of policies. It ensures that their semantics is not affected by low-level implementation choices made in the development of the MAS.

Other work, e.g. [145] has used Lamport's Temporal Logic of Actions (TLA) [82] to express policies. Although TLA is a suitable logic to represent the execution of programs we preferred ITL in this work as it provides a intuitive operator for sequential composition (Chop) and iteration (Chopstar) and also allows for the specification of intervals at different granularities of time (Temporal Projection). The sequence of actions can also be expressed using TLA, but the specification would not be as concise as in ITL. As this work uses the above mentioned concepts frequently, ITL represented a more suitable choice. The advantage of a formalism like TLA is that the notion of atomic actions is a fundamental part of the logic itself. This may have been more suitable for the modelling of agent actions, however since the main focus of the thesis is the integration of security and functionality we feel that the advantages of the sequential composition and temporal projection outweigh TLA's benefit of atomic actions.

4.2 Syntax and Semantics

The key notion of ITL is an *interval*. An interval σ is considered to be a (in)finite sequence of states σ_0, $\sigma_1 \ldots$, where a state σ_i is a mapping from the set of variables Var to the set of integer values $\mathbb{Z}$. The length $|\sigma|$ of an interval $\sigma_0 \ldots \sigma_n$ is equal to n (one less than the number of states in the interval, so a one state interval has length 0. This has always been a convention in ITL).

Expressions
$e ::= \quad \mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \bigcirc v \mid \mathsf{fin}\ v$
Formulae
$f ::= \quad p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2 \mid f^*$

Table 4.1: Syntax of ITL

The syntax of ITL is defined in Figure 4.1 where μ is an integer value, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol. The informal semantics of the most interesting constructs are as follows:

- skip: unit interval (length 1, i.e., an interval of two states).
- $f_1 ; f_2$: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval. Note the last state of the interval over which f_1 holds (highlighted in red) is shared with the interval over which f_2 holds. This is illustrated in Figure 4.1.

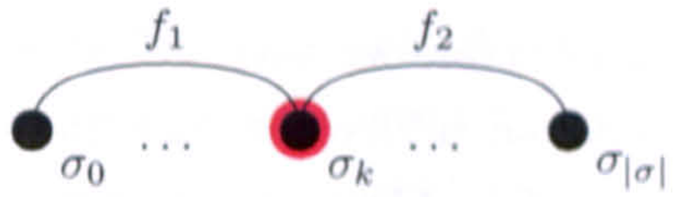


Figure 4.1: Informal Semantics of $f_1 ; f_2$

- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds. This is illustrated in Figure 4.2.

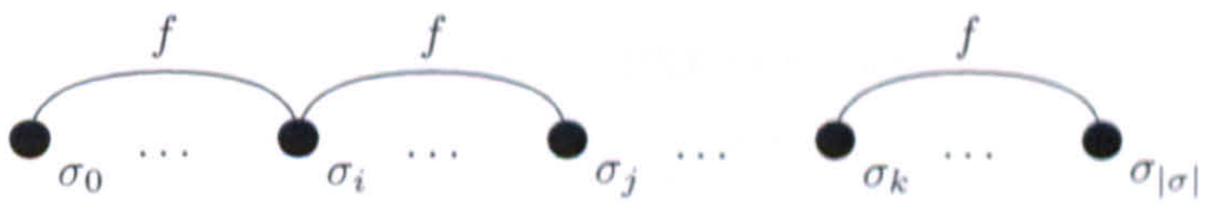


Figure 4.2: Informal Semantics of f^*

- $\bigcirc v$: value of v in the next state when evaluated on an interval of length at least one, otherwise an arbitrary value.
- fin v: value of v in the final state when evaluated on a finite interval, otherwise an arbitrary value.

4.3 Formal Semantics

The meaning of terms and formulae is defined in this section. We are only interested in the functions and relations over integer numbers. Let $\mathbb{Z}$ stand for the set of integer numbers, and *Var* the set of integer variables. In the sequel, we denote by $E \to F$ the set of all *total* functions from E to F.

We assume that a total function $\hat{g} \in \mathbb{Z}^n \to \mathbb{Z}$ is associated with each n-ary function symbol g, and a total function $\hat{p} \in \mathbb{Z}^n \to \{\mathrm{tt}, \mathrm{ff}\}$ is associated with each n-ary relation symbol p. Function symbols, e.g. $+$ and $-$, and relation symbols, e.g. $\geq$ and $=$, are assumed to have their standard meanings. In particular, the truth-values tt and ff are associated with *true* and *false*, respectively.

Expressions and formulae are evaluated over intervals. Remember that an interval is defined to be a (in)finite nonempty sequence of states $\sigma \mathrel{\widehat{=}} \sigma_0\sigma_1\ldots$, where each state σ_i is a *value assignment* which associates an integer number with each variable:

$$\sigma_i \in \Sigma \mathrel{\widehat{=}} Var \to \mathbb{Z}.$$

We denote by Σ^+ and Σ^ω the sets of finite intervals and the set of infinite intervals respectively. If σ is an infinite interval, we take $|\sigma| = \infty$ and write $\sigma = \sigma_0\sigma_1\ldots\sigma_\infty$. Furthermore, for any $i, j \in \mathbb{N}$ and an interval σ such that $i \leq j \leq |\sigma|$, we write $\sigma[i, j]$ to denote the subinterval $\sigma_i\ldots\sigma_j$ of σ. Given two intervals $\sigma, \sigma' \in (\Sigma^+ \cup \Sigma^\omega)$, we write $\sigma \sim_v \sigma'$ if the intervals σ and σ' are identical with the possible exception of their mappings for the variable v, i.e $|\sigma| = |\sigma'|$ and $v \neq v' \Rightarrow \sigma_i(v') = \sigma'_i(v')$ for $i = 0, 1, \ldots, |\sigma|$.

The *semantics of an expression* e is a function

$$\mathcal{E}[\![e]\!] \in (\Sigma^+ \cup \Sigma^\omega) \to \mathbb{Z},$$

defined inductively on the structure of expressions by

$$\begin{aligned}
\mathcal{E}[\![v]\!](\sigma) &= \sigma_0(v) \\
\mathcal{E}[\![g(e_1, \ldots, e_n)]\!](\sigma) &= \hat{g}(\mathcal{E}[\![e_1]\!](\sigma), \ldots, \mathcal{E}[\![e_n]\!](\sigma)) \\
\mathcal{E}[\![\bigcirc v]\!](\sigma) &= \begin{cases} \sigma_1(v) & \text{if} \quad |\sigma| > 0 \\ \chi(\mathbb{Z}) & \text{otherwise} \end{cases} \\
\mathcal{E}[\![\,\mathsf{fin}\ v]\!](\sigma) &= \begin{cases} \sigma_{|\sigma|}(v) & \text{if } \sigma \text{ is finite} \\ \chi(\mathbb{Z}) & \text{otherwise} \end{cases}
\end{aligned}$$

where χ denotes a choice function which maps any nonempty set to some element in the

3.5.4 Enforcement Mechanism in SANTA

Enforcement Mechanisms in SANTA represent the association of a policy with the concrete mechanism for the enforcement. This association allows to check the constraints on the policies to be enforceable by the mechanism statically. The following listing provides three examples of how a policy can be associated with a vigilant agent, vigilant object, or a security enforcer.

Listing 3.4: Examples of Enforcement Mechanisms

```
1  /* Vigilant Object */
2  enforce pid with oid
3  /* Vigilant Agent */
4  enforce pid with aid
5
6  /* Security Enforcer */
7  securityenforcer SE :
8     protect (oid1, oid2 /* , ... */)
9  end
10
11 enforce pid with SE
```

Line 2 states that the policy with the identifier `pid` is enforced by the enforcement mechanism implemented in the vigilant object `oid`. Similarly Line 4 states that the policy `pid` is enforced in the vigilant agent `aid`. Line 7 to 9 define a security enforcer with the name `SE` that controls access to the objects `oid1` and `oid2`. Line 11 then associates the policy identified by `pid` with the security enforcer `SE`.

3.6 Summary

This chapter provided a brief overview of the computational model and the SANTA design-language. It described the components that define the behaviour of the SMAS:

- *Agents* are active and execute sequentially *local* or *remote actions* (computation, respectively synchronous communication). The execution model consists of three main phases: a) Deliberation: Prioritisation of actions; b) Enforcement: Constraining actions for execution; and c) Execution: Execution of the selected action.
- *Objects* are passive and can be accessed through defined interfaces. They are protected by either vigilant or security enforcer mechanisms. The access to objects is mutually exclusive.
- *Policies* define constraints on the execution of the SMAS. They are enforced by enforcement mechanisms. Policies can dynamically change over time and on events.

set. The *semantics of a formula* f is a function

$$\mathcal{M}[\![f]\!] \in (\Sigma^+ \cup \Sigma^\omega) \to \{\mathrm{tt}, \mathrm{ff}\},$$

defined inductively on the structure of formulae below, where the following abbreviation is used:

$$\sigma \models f \;\widehat{=}\; \mathcal{M}[\![f]\!](\sigma) = \mathrm{tt}$$
$$\sigma \not\models f \;\widehat{=}\; \mathcal{M}[\![f]\!](\sigma) = \mathrm{ff}$$

The definition of $\mathcal{M}[\![f]\!]$ is

$\sigma \models p(e_1, \ldots, e_n)$	iff	$\hat{p}(\mathcal{E}[\![e_1]\!](\sigma), \ldots, \mathcal{E}[\![e_n]\!](\sigma))$
$\sigma \models \neg f$	iff	$\sigma \not\models f$
$\sigma \models f_1 \wedge f_2$	iff	$\sigma \models f_1$ and $\sigma \models f_2$
$\sigma \models \forall v \bullet f$	iff	$\sigma' \models f$, for all σ' such that $\sigma \sim_v \sigma'$
$\sigma \models \mathsf{skip}$	iff	$\lvert\sigma\rvert = 1$
$\sigma \models f_1 \,;\, f_2$	iff	$(\sigma[0,k] \models f_1$ and $\sigma[k, \lvert\sigma\rvert] \models f_2$, for some $k \in \mathbb{N},\ 0 \leq k \leq \lvert\sigma\rvert)$ or $(\lvert\sigma\rvert = \infty$ and $\sigma \models f_1)$
$\sigma \models f^*$	iff	(exist $l_0, \ldots, l_n \in \mathbb{N}$ such that $l_0 = 0 \leq \ldots \leq l_n = \lvert\sigma\rvert$ and $\sigma[l_i, l_{i+1}] \models f, 0 \leq i \leq n-1)$ or (exist $l_0, \ldots, l_\infty \in \mathbb{N}$ such that $l_0 = 0$ and $l_i \leq l_{i+1}$ and $\sigma[l_i, l_{i+1}] \models f,\ i \in \mathbb{N}$)

A formula f is *valid*, written $\models f$ iff $\sigma \models f$ for every interval $\sigma \in (\Sigma^+ \cup \Sigma^\omega)$. A formula f is *satisfiable* iff $\sigma \models f$ for some interval σ. ITL has got a sound and compositional proof system. Interested readers are referred to [40, 101] for the proof system and further details about the logic.

4.4 Derived constructs

The following is a list of some derived constructs which are useful for the specification of systems or policy rules:

$\bigcirc f \mathrel{\widehat{=}} \mathsf{skip} \,;\, f$	next f, f holds from the next state. Example: $\bigcirc X = 1$: Any interval such that the value of X in the second state is 1 and the length of that interval is at least 1.
$\mathsf{more} \mathrel{\widehat{=}} \bigcirc\mathsf{true}$	non-empty interval, i.e., any interval of length at least one.

$\mathsf{empty} \mathrel{\widehat{=}} \neg\, \mathsf{more}$	interval, i.e., any interval of length zero (just one state).
$\mathsf{inf} \mathrel{\widehat{=}} \mathsf{true}\,;\mathsf{false}$	infinite interval, i.e., any interval of infinite length.
$\mathsf{finite} \mathrel{\widehat{=}} \neg\, \mathsf{inf}$	finite interval, i.e., any interval of finite length.
$\Diamond f \mathrel{\widehat{=}} \mathsf{finite}\,;f$	sometimes f, i.e., any interval such that f holds over a suffix of that interval. Example: $\Diamond X \neq 1$: Any interval such that there exists a state in which X is not equal to 1.
$\Box f \mathrel{\widehat{=}} \neg \Diamond \neg f$	always f, i.e., any interval such that f holds for all suffixes of that interval. Example: $\Box(X = 1)$: Any interval such that the value of X is equal to 1 in all states of that interval.
$\diamondsuit f \mathrel{\widehat{=}} f\,;\mathsf{true}$	diamond-i, i.e., any interval such that f holds over a prefix sub-interval.
$\boxdot f \mathrel{\widehat{=}} \neg \diamondsuit \neg f$	box-i, i.e., any interval such that f holds over all prefix sub-intervals.
$\blacklozenge f \mathrel{\widehat{=}} \Diamond(\diamondsuit f)$	diamond-a, i.e., any interval such that f holds over a sub-interval.
$\blacksquare f \mathrel{\widehat{=}} \neg \blacklozenge(\neg f)$	box-a, i.e., any interval such that f holds over all sub-intervals.
$\mathsf{keep}\ f \mathrel{\widehat{=}} \blacksquare\,(\mathsf{skip} \supset f)$	keep f, i.e., any interval such that f holds over all unit sub-intervals.
$\mathsf{halt}\ f \mathrel{\widehat{=}} \Box(\mathsf{empty} \equiv f)$	terminate interval when f holds.
$\mathsf{fin}\ f \mathrel{\widehat{=}} \Box(\mathsf{empty} \supset f)$	final state, i.e., any interval such that f holds in the final state of that interval.
$v := e \mathrel{\widehat{=}} (\bigcirc v) = e$	assignment, i.e., the value of v will be e in the next state.
$v \leftarrow e \mathrel{\widehat{=}} \mathsf{finite} \wedge (\mathsf{fin}\ v) = e$	temporal assignment, i.e., the value of v in the final state will be the value of e.
$\mathsf{stable}\ v \mathrel{\widehat{=}} \Box(\mathsf{more} \supset v := v)$	remain stable, i.e., the value of v remains stable in the interval.
$v\ \mathsf{gets}\ e \mathrel{\widehat{=}} \Box(\mathsf{more} \supset v := e)$	gets, i.e., in every state except the initial state the variable v will be assigned the value of e evaluated in the previous state.
$\mathsf{len}(e) \mathrel{\widehat{=}} \exists I \bullet (I = 0) \wedge (I\ \mathsf{gets}\ I + 1) \wedge \mathsf{fin}\,(I = e)$	holds if the length of the interval is e.

4.5 Operator Always-followed-by

As most other policy models, our policy model uses rules as the smallest unit of specification. The semantics of the operator has been introduced by Siewe [124] using the operator *always-followed-by*, denoted by the symbol " $\mapsto$ ". The operator is defined as follows:

$$f \mapsto w \mathrel{\widehat{=}} \boxdot ((\Diamond f) \supset \mathsf{fin}\ w) \tag{4.1}$$

where f stands for any ITL formula, and w is a state formula (an ITL formula without temporal constructs). The definition (4.1) states that whenever the formula f holds in a (finite) sub-interval then the state formula w must hold in the final state of that sub-interval. That is, f is always followed by w. The informal meaning of the operator is portrayed in Figure 4.3.

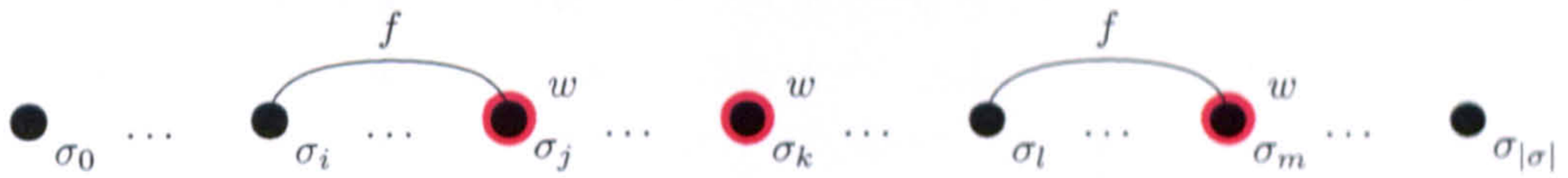


Figure 4.3: Informal Semantics of $f \mapsto w$

Note that in (4.1), w can be true in a state even if f does not hold in the left neighbourhood of the state. This operator will be used subsequently in Chapter 7 to express policy rules. For a discussion of useful properties of the operator we refer the reader to [124].

4.6 Temporal Projection

To be able to express the specification of policies and enforcement at a higher level of abstraction we use in this work the temporal projection operator that has been presented by Moszkowski in [100]. Temporal projection allows to relate specifications that are defined on a different granularity of time. It is used in this work to define policies and their enforcement mechanisms in abstraction of their concrete implementations. The following informal definition is adapted from [100].

Roughly speaking, the formula $S\Delta T$ is defined to be true on an interval σ iff two conditions are met. First, the formula T must be true on some interval σ' obtained by projecting some states from σ. Second, the formula S must be true on each of the subintervals of σ bridging the gaps between the projected states.

The semantics of $S\Delta T$ is formally expressed as:

$$\begin{aligned} \sigma \models S\Delta T \quad \text{iff} \quad & \text{for some } n \geq 0, \sigma' \text{ and } l_0, \ldots, l_n : \\ & 0 = l_0 < \cdots < l_n = |\sigma|, \text{ and for each } i < n, \quad \sigma[l_i, l_{i+1}] \models S, \\ & \text{and } \sigma' \models T \text{ where } |\sigma'| = n \text{ and for all } i \leq n, \quad \sigma'_i = \sigma_{l_i} \end{aligned}$$

It can be shown that the operator f^* can be expressed in terms of the projection operator as $f\Delta\mathsf{true}$. Moszkowski also provides a simple example that shows the application of the projection operator. The example is depicted in Figure 4.4.

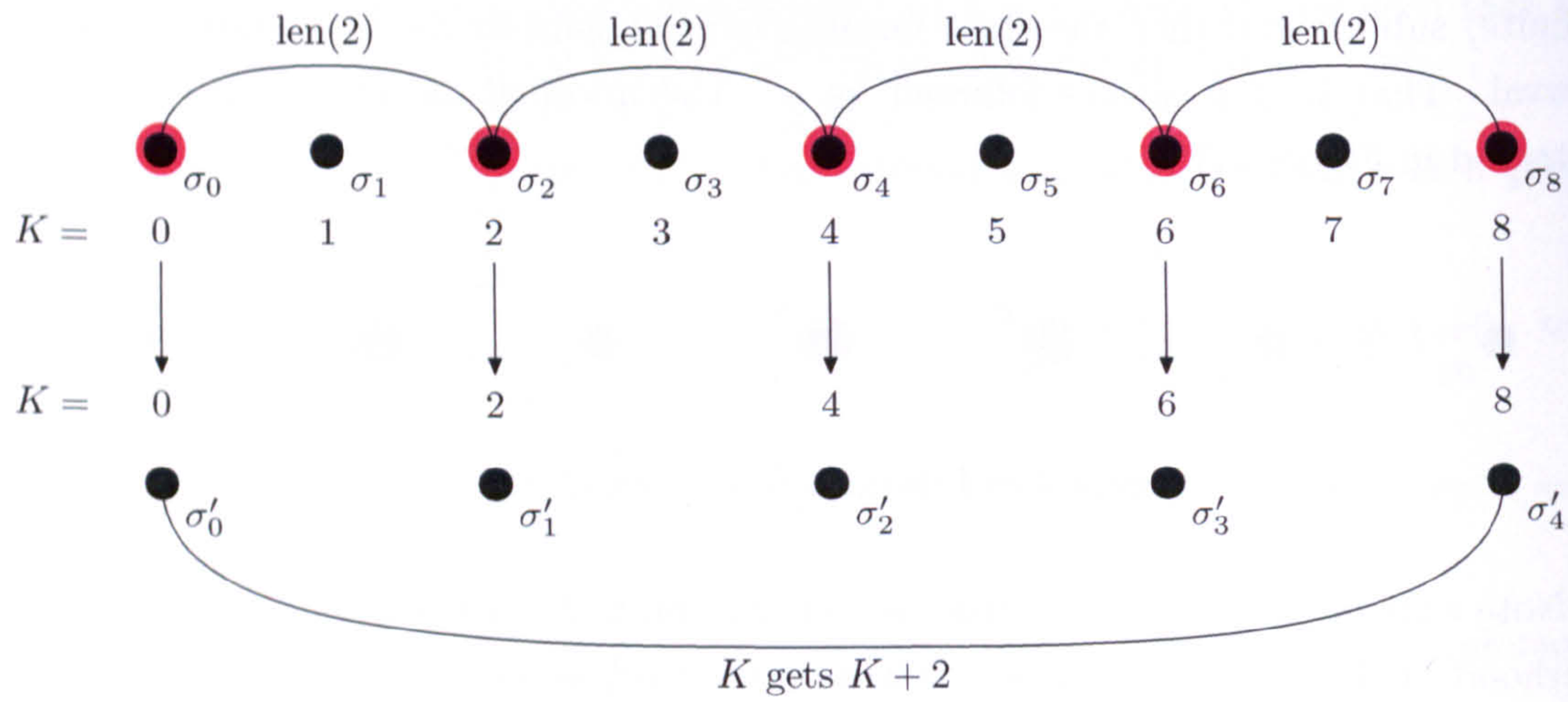


Figure 4.4: Example of Temporal Projection

In the interval σ the value of K increases from 0 to 8 in steps of one. The interval σ satisfies $(\mathsf{len}(2))\Delta(K \text{ gets } K + 2)$. $(\mathsf{len}(2))$ is true if the interval is of length two and $(K \text{ gets } K + 2)$ is true if the K increases by 2 from state to state. The gaps between the projected states (highlighted in red) are bridged by the formula $\mathsf{len}(2)$. This shows how specifications that are defined at different levels of temporal granularity can be mapped. We use temporal projection in Chapter 8 to define the mapping between policies and system specification. More detail on projection and its properties can be found in [100].

Chapter 5

Agents

We formalise the behaviour of independent agents that perform only local computations on their encapsulated variables. The focus is on the execution model of an individual agent, and the interaction between the initialisation, deliberation, enforcement, execution and termination phases.

5.1 Introduction

This chapter provides the specification oriented semantics of a subset of the SANTA design language. The subset, referred to as $SMAS_1$, consists of a set of agents, that perform local computations independently from each other. Communication using shared objects, policies and their enforcement mechanisms are introduced subsequently in Chapters 6 to 8. The motivation to split the description of SANTA's denotational semantics in different chapters originates from its complexity.

The focus of this chapter is the execution model of a single SANTA agent and the semantics of the *initialisation* phase, the *deliberation-enforcement-execution* cycle, and the *termination* phase. The syntactic constructs of the language that allow to define agents are formally defined and their semantics is described both informally and formally as ITL formulae.

The chapter is structured as follows. In Section 5.2 the SANTA syntactic constructs for the specification of agents and their informal semantics is given, along with a small motivating example. Section 5.3 formalises the definitions provided in the previous section, yielding the semantics of a single agent. Section 5.4 combines the definitions to yield the semantics of the $SMAS_1$ that consists of a set of agents. This is illustrated in Section 5.5 where the motivating example, that was given in Section 5.2.4, is discussed formally. Finally, Section 5.6 concludes this chapter with a summary.

5.2 Agents in SANTA

The syntax of a $SMAS_1$ is given in EBNF as described in the ISO/IEC 14977:1996(E) [70] in Listing 5.1. Terminal strings are character sequences enclosed in single quotes. Meta-identifiers are defined using the defining symbol "=". Concatenation is expressed by a comma. Optional sequences are enclosed in square brackets; repetition sequences are enclosed in braces. Parentheses are used for grouping. Each declaration is terminated by a full stop and the vertical bar depicts a choice.

The definitions for expressions (`expr`), identifiers (`id`) and literals (`lit`) are omitted and assumed to follow standard conventions. Expressions are *side-effect free*.

Given this syntax the system is specified as a set of agents, in the following denoted by $\mathcal{A}$. Each agent definition $a \in \mathcal{A}$ is introduced by the keyword `agent` followed by its system wide unique identifier id_a, that is separated from the agent's variable definitions by a colon. The agent definition is closed by the keyword `end`. An agent optionally defines a set of variables and actions and exactly one *deliberation* section.

Listing 5.1: EBNF for $SMAS_1$

```
1  MAS       = { agent } .
2
3  agent     = 'agent', id, ':', {vardecl}, {actdecl}, delib, 'end'.
4
5  vardecl   = 'var', id, '=', lit.
6
7  actdecl   = 'when', expr, 'do', id, ':', statement.
8
9  statement = 'idle'
10           | ( id, { ',', id } ':=', expr, { expr, ',' } )
11           | ( id, { ',', id } '←', expr, { expr, ',' })
12           | ( statement, ';', statement )
13           | ( 'if', expr, 'then', statement, ['else', statement])
14           | ( 'for', id, '<', expr, 'do', statement )
15           | ( 'var' id, { ',', id }, ←', lit, {',' lit }, ':' statement
                )
16           | ( 'terminate' )
17           | ( '{', statement, '}' ).
18
19 delib     = 'deliberation', ':',
20             (( '{', {vardecl}, statement, '}') | 'external').
```

5.2.1 Agent State Variables

All variables must have an agent-wide unique identifier and are assigned an initial value. Variables can be declared at the agent level (*Agent Variables*, denoted by $V_{a,decl}$), at the action level (*Local Variables*) and in the deliberation section (*Deliberation Auxiliary Variables*, denoted by $V_{a,delib}$). Auxiliary Variables required for the enforcement (*Enforcement Auxiliary Variables*, denoted by $V_{a,enf}$) are implicitly defined by vigilant enforcement mechanisms. The *Agent Control Variables*, denoted by $V_{a,contr}$, are defined at the semantic level. The set of all *Agent State Variables*, denoted by V_a is the union:

$$V_a = V_{a,decl} \cup V_{a,delib} \cup V_{a,enf} \cup V_{a,contr}$$

In the semantics all variables in the set V_a are modelled as ITL state variables. This is important to keep in mind, as the SANTA syntax does allow to define lower-case variable names. Although not following the ITL notational convention these are ITL state variables. As all variables are state variables the behaviour of the agent is required to define the value of variables for every state, typically leading to large specifications. To keep the specification small, the statement semantics is defined with respect to a set of variables. The intuition is that the statement semantics defines the behaviour of *all* variables in this set in such a way that the variables can be thought of as being *memory variables*. Memory variables, sometimes also called program variables, are persistent and maintain their previous values, unless they are explicitly assigned. The concept of memory variables

is introduced for the convenience of the specification, as ITL state variables do not by default maintain their values. A complete ITL specification must define their values for each state. Memory variables are formally defined in Section 5.3.3.

Accessibility of Agent State Variables

All *Agent State Variables* are encapsulated in the agent and are not accessible by other agents or objects. Table 5.1 provides an overview of the accessibility of the agent's state variables during the different phases of an agent's execution.

	Symbol	Deliberation	Enforcement	Execution
Agent Variables	$V_{a,decl}$	read	read	read/write
Deliberation Aux. Variables	$V_{a,delib}$	read/write		
Enforcement Aux. Variables	$V_{a,enf}$		read/write	
Agent Control Variables	$V_{a,contr}$			
done($id_{a,x}$)	$done_{a,x}$	read	read	read
failed($id_{a,x}$)	$failed_{a,x}$	read	read	read
Priority Variables	$\Pi_{a,x}$	write	read/write	
Termination	$doterm_a$	write	write	write
Others	$ready_a, term_a$			

Table 5.1: Accessibility of Agent State Variables

Agent Variables are local to the agent definition and may be read in any expression that is used in the agent definition. They may only be assigned in the statement of actions that are defined in this agent. *Local Variables* are declared as part of the statement syntax and can only be read and assigned in the subsequent statement. *Deliberation Auxiliary Variables* are declared at the beginning of the deliberation section. They are local to this phase and can only be read and assigned by the statements in this phase.

Some of the *Agent Control Variables* are accessible using predefined constructs. The Boolean variables $done_{a,x}$ and $failed_{a,x}$ denote the success or failure of agent a's action x. They are accessible using the done($id_{a,x}$) and failed($id_{a,x}$) constructs, where $id_{a,x}$ denotes the identifier of the agent's action x. The *priority variables* $\Pi_{a,x}$ denote the priority with which the agent a's action x is to be executed. They are accessible in the syntax by the identifier of the action $id_{a,x}$. To allow for the termination of a single agent the boolean control variable $doterm_a$ is introduced. Its value is initialised to *false* and can only be set to *true* using the dedicated statement **terminate** in the statement syntax. The constraints, listed in Table 5.1, on the accessibility of the variables can be statically checked.

5.2.2 Agent Capabilities

An action definition is introduced by the keyword `when` that is followed by a Boolean expression representing the precondition of the action. The precondition of the agent a's action x is denoted by $p_{a,x}$. Separated from the precondition by the keyword `do` is the agent wide unique identifier of the action. The identifier of agent a's action x is in the following denoted by $id_{a,x}$. It is used in policy specifications to constrain the agent's behaviour and in the syntax of the deliberation section to reference the action's priority variable. Introduced by a colon follows the statement that defines the behaviour of the agent when executing the action. The statement of agent a's action x is denoted in the following by $S_{a,x}$. The action definition is closed by a full-stop.

Precondition

The precondition $p_{a,x}$ is a constraint describing a *necessary* condition for the action's execution. It should not reflect the choice of the agent, as this is encoded in the deliberation phase and the policies that constrain the behaviour, but rather conditions under which the action can possibly be executed. For example:

An agent that is controlling a robot arm, implements an action `droppayload`, that drops the payload to a predefined destination. A precondition for this action would be that the arm is currently holding some payload, as it is otherwise impossible to drop it. However, the fact that the agent is holding a payload does not imply that it will drop it.

An action may only be executed if its precondition is *true*. This makes the precondition a static design decision that *cannot* be overruled by decisions made in the deliberation or enforcement phase. As a consequence obligations cannot be enforced if the agent's state does not satisfy the precondition. Care should be taken in the agent design to keep the constraints expressed by preconditions limited, as this enhances the maintainability of the agent.

We currently assume that the deliberation process of the agent is not aware of the policy decisions. Consequently the agent will decide on the execution of an action using for example the action's utility. If the chosen action was not permitted the agent is informed of the failure by the appropriate settings of the control variables in the next iteration of the deliberation phase. To minimise performance degradation the agent can prioritise actions as opposed to choosing exactly one action. If the highest priority action is not permitted the execution defaults to the action with the second highest priority. We believe that this approach outweighs the complexity of computing the access rights for all agent actions before the deliberation step.

Statement

The statement syntax is used to define the computation steps of the agent when executing the action and also to specify the deliberation phase. The commonly used procedural constructs are available. Additionally the *temporal multiple assignment* is provided as a specification construct. The statements have the following meaning:

The Idle Statement denotes a unit interval wait, during which all *Agent State Variables* maintain their values.

Concrete Multiple Assignment assigns a list of values to a list of variables concurrently. The assignment of all values is performed within one unit interval. As this may not always be implementable the use of the *Temporal Multiple Assignment* is encouraged for the specification. Constraints on the statement are that the number of identifiers on the left hand side of the `:=` operator matches the number of expressions on the right hand side. Additionally each identifier may only occur once on the left-hand side of the operator. An assignment of a single variable over a unit interval represents a special case of the concrete multiple assignment.

An example of the use is a swap statement that exchanges the values of two variables: `a,b := b,a`, *where a and b are Agent Variables.*

Temporal Multiple Assignment assigns a list of values to a list of variables. It is provided as a specification construct, as it does not specify the time in which the assignment completes. The time is only assumed to be *finite*. This allows for the later refinement to take advantage of the target execution platform's capabilities. The same constraints as for the concrete multiple assignment apply.

The temporal multiple assignment: `a,b ← b,a` *can be for example be implemented as a concrete multiple assignment:* `a,b := b,a` *or for target platforms that do not support concurrent assignment, as the sequence:* `var t ← a :{ a := b ; b := t }`

Sequence The sequential composition of two statements is denoted by `S1 ; S2`.

Conditional Choice is denoted by `if expr then S1 else S2`. The expression `expr` must evaluate to a Boolean value. If expr is true then S1 is executed, otherwise S2. The `else S2` part is optional.

Iteration is denoted by the `for id < expr do S` construct, where *id* is the loop variable that starts with 0 and *expr* must evaluate to an integer. The loop statement is executed

and the loop variable incremented by one as long as it is lower than the specified limit. The limit is evaluated once, before the first execution of the statement *S*. The statement may not assign any value to the loop variable. This can be statically checked.

Local variables are introduced using the `var v ← expr :S` construct. The variable identifiers must be unique within the agent definition. The scope of the variables is limited to the subsequent statement. The initialisation of local variables is defined using the *Temporal Multiple Assignment*, allowing to refine the time that the variable allocation and initialisation take with respect to the target platform.

Termination of the agent is indicated by the statement `terminate`. Informally it means that the agent will terminate at the end of the current deliberation-enforcement-execution cycle.

Expressions All expressions are assumed to be free of side-effects, viz. the assignment is strictly a statement and the evaluation of any expression is assumed to be instantaneous. The expression syntax is not detailed here, as this would not provide any additional insights into the behaviour of the system and the enforcement of policies. However, Listing 5.2 provides a quick overview of some of the expressions that are allowed. We assume that a primitive type system can be implemented, that supports boolean, integer and lists.

The common arithmetic operators are defined for integers. The + operator is overloaded and can also be used to concatenate lists. The conditional expression is similar to the ternary operator in procedural programming languages. Relational operators can be used for integers. Equality and inequality can be also used to compare boolean and list expressions.

List expressions are in order of occurrence: list dereference, yields the element of the list the specified index; sublist, yields the sublist from a given start index to a given end index (both inclusive); The length of the list is obtained using the bars. Universal and existential quantification can be used to quantify over lists. It is possible to define functions as part of the agents, objects and policies. These definitions can be used locally in expressions.

There are several reserved keywords in the language. Their usage is often restricted to a specific context, in which the expression can be used. The identifiers `list`, `int` and `bool` denote the different types that are available. The identifier `T` is used in the specification of policy rules to reference the time that elapsed since the rule started to be enforced. The identifier `any` is used in the policy scope definitions, to denote that the universal set of subjects, objects or actions is to be used. The predefined functions `done` and `failed` are

used to access the respective control variables that indicate the success or failure of the last executed action.

Listing 5.2: EBNF for Expressions

```
1  expr  =  /* Arithmetic */
2     expr, '%', expr | expr, '*', expr | expr, '/', expr |
3     expr, '+', expr | expr, '-', expr |
4
5     /* Conditional */
6     'if', expr, 'then', expr, [ 'else', expr ] |
7
8     /* Relational */
9     expr, '<', expr  | expr, '<=', expr | expr, '>', expr |
10    expr, '>=', expr | expr, '=', expr  | expr, '<>', expr |
11
12    /* Boolean  */
13    'not', expr | expr, 'and', expr | expr, 'or', expr |
14    'type', '(', expr, expr, ')' |
15
16    /* List */
17    expr, '[', expr, ']' | expr, '[', expr, '..', expr, ']' |
18    '|', expr, '|' |
19    ( 'forall' | 'exists' ), id, '<', expr, ':', expr |
20
21    /* Function */
22    id, '(', [ expr, { ',', expr }, ] ')' |
23
24    /* Reserved  */
25    'list' | 'int' | 'bool' | 'T' | 'any' |
26    'done', '(',id,')' | 'failed', '(',id,')' |
27    'done', '(',id,',',id,',',id, ['(',id, {',',id},') '],')' |
28    'failed', '(',id,',',id,',',id, ['(',id,{ ',',id },') '],') '|
29
30    /* Identifier, Literals */
31    id | lit | listlit .
```

The anchor of the expression definition is an identifier, an integer or boolean literal, or a list literal. List literals are an enumeration of expressions of the form: `'[', [expr, {',', expr}], ']'`.

5.2.3 Agent Deliberation

The deliberation phase abstracts from the complex decision making processes of (*intelligent*) agents. The deliberation phase is loosely coupled with the execution of the agent, using the priority variables as an interface. As a result the deliberation phase can be seen as a distinct module within the agent, that decides on the agent's course of action by prioritising its capabilities. The SANTA design-language allows us to leave the deliberation phase at an abstract level using the keyword **external**. This means that the agent assigns non-deterministically a non-negative integer value as the priority for each of its actions.

Alternatively the deliberation phase can be explicitly defined using *deliberation auxiliary variables* and the statement syntax. The observable variables in the statement are all *Agent Variables* and *Deliberation Auxiliary Variables*. Additionally the Boolean *Control Variables* that indicate the success or failure of the last action can be accessed using the special language constructs: `done(id)` and `failed(id)`, where *id* is equal to $id_{a,x}$, the name of one of the agent's actions. The access to the control variables provides feedback of the success or failure of the previously executed action to the agent's deliberation phase. This gives the agent the opportunity to adequately respond to security failures. Access to this variables is also important in the specification of history dependent policies.

Variables that can be assigned in the statement of the deliberation section are the *Deliberation Auxiliary Variables* and the *Priority Variables* of the agent's actions. The latter are syntactically represented as variables with the identifier of the corresponding action. The range that can be assigned to priority variables is restricted to non-negative integer values. The value 0 has the special meaning that the corresponding action should not be executed; 1 denotes the highest priority and greater values denote lower priorities. Theoretically the lowest priority is unbounded, however any implementation will have to define a lower bound for priorities, which is denoted by $\bot_\Pi$[1].

The choice to encode the priority values as integers is arbitrary. Any total ordering reflecting the priority of actions can be used, provided it defines a single element (0) that indicates that the action must not be executed. The ordering must satisfy that $\top_\Pi \leq 0 \leq \bot_\Pi$. In the following we assume that the priority ordering defined as integers, with the integer value 0 as the middle element, $\top_\Pi = \texttt{MIN}_{\texttt{INT}}$ and $\bot_\Pi = \texttt{MAX}_{\texttt{INT}}$. The concrete values of $\texttt{MIN}_{\texttt{INT}}$ and $\texttt{MAX}_{\texttt{INT}}$ depend on the actual implementation platform.

5.2.4 Motivating Example

Assume a simplistic system consisting of a single agent *A* that continuously increments its encapsulated variable *x* if it is even or doubles it if its current value is odd. Listing 5.3 shows the SANTA agent specification for this example.

The agent *A* encapsulates a single *Agent Variable* *x* that is initialised to *0*. It defines two internal actions named *inc* and *dbl*. The precondition for *inc* is that the current value of *x* is less than $2^{32} - 1$ and the precondition for *dbl* is that the current value of *x* is less than 2^{31}. In this case the preconditions are used to prevent an integer overflow in case that *x* represents an 32-bit unsigned integer. The statements of the actions define the respective computation on the variable *x*.

[1]The bounds $\top_\Pi$ and $\bot_\Pi$ should be read as the *top* priority and the *least* priority. The corresponding integer values are exactly reversed.

Listing 5.3: Example Agent Specification

```
1  /* Agent Definition */
2  agent ag :
3
4      /* Agent Variables */
5      var x = 0
6
7      /* Agent Actions */
8      when x < 2^32 - 1 do inc : x := x+1
9      when x < 2^31 do dbl : x := 2*x
10
11     /* Deliberation */
12     deliberation : {
13     if x % 2 = 0 then inc,dbl ← 1,0
14                  else inc,dbl ← 0,1 }
15 end
```

The deliberation is in this example explicitly specified and implements the decision to give *inc* preference if x is even, viz. x modulo 2 is zero, and otherwise give preference to the action *dbl*. This example is discussed in greater detail in Section 5.5, where the use of the control variables that are needed to give the formal semantics of the $SMAS_1$ system are illustrated.

The provided example is kept simple, deliberately. They illustrate the formal semantics of the SANTA language constructs, described in the subsequent sections. The specification is used to provide a concise definition of the system behaviour to show the interaction between the system and the enforced policies. This interaction is not influenced by the amount of action definitions nor the complexity of the action or deliberation statements.

5.3 Single Agent Semantics

The system consists of a set of agents $\mathcal{A}$ of independent sequential processes that are executing concurrently. To control that the agent does only execute one action at a time, an additional control variable is introduced to capture the readiness of an agent to execute an action. This is especially important in the subsequent chapters that introduce communication.

Agent Readiness An agent as a sequential process may only execute one action at a time. In order to capture this requirement we introduce the additional control variables $ready_a \in V_{a,contr}$, where $a \in \mathcal{A}$. $ready_a$ is a boolean ITL state variable; its behaviour is specified by the semantics of the agent. $ready_a = true$ indicates that the agent is *ready* to execute an action.

In the following the different phases of an agent's execution cycle are formalised.

5.3.1 Initialisation Phase

Before an agent enters its *deliberation-enforcement-execution* cycle, it initialises its agent state variables. The initialisation is defined in (5.1).

$$\begin{aligned} init_a \mathrel{\widehat{=}} & \; ready_a = false \wedge (\mathsf{keep}\ ready_a = false) \;\wedge doterm_a, term_a \leftarrow false, false \;\wedge \\ & \bigwedge_{x \in X_a} (done_{a,x} \leftarrow false \wedge failed_{a,x} \leftarrow false \wedge \Pi_{a,x} \leftarrow 1) \;\wedge \\ & \bigwedge_{v \in V_{a,decl}} (v \leftarrow I_v) \end{aligned} \tag{5.1}$$

The initialisation of an agent must ensure that the agents control variable $ready_a$ is *false* throughout the initialisation. This reflects that the agent cannot engage in the execution of an action during initialisation. The initialisation also ensures that when the agent enters its deliberation-enforcement-execution cycle the control variables $doterm_a$ and $term_a$ are set to *false*. This is expressed in the first line of (5.1).

Additionally the boolean control variables $done_{a,x}$ and $failed_{a,x}$ are initialised to the value *false*, reflecting that no action has been executed, yet. The priority variables are all assigned to the *same* value; the choice of the value 1 is arbitrary. This is expressed in the second line of (5.1).

The last line of (5.1) denotes the initialisation of the agent variables to their initial values. Here I_v denotes the initialisation value that has been specified in the agent definition. After the initialisation the agent enters the deliberation phase.

5.3.2 Deliberation Phase

The agent enters its deliberation-enforcement-action cycle at the deliberation phase. In the SANTA language, this phase is either left unspecified using the key word **external** or explicitly defined by deliberation auxiliary variables and a statement that defines the agent's behaviour during the execution of this phase. The semantics of both cases is provided in the following.

External Definition

The keyword **external** denotes that the priority variables $\Pi_{a,x}$ of the agent a's actions $x \in X_a$ take any non-negative integer value in the final state of a finite interval. The

semantics of **deliberation : external** is:

$$delib_{a,external} \;\widehat{=}\; \mathsf{finite} \wedge \bigwedge_{x \in X_a} \mathsf{fin}\,(\Pi_{a,x} \geq 0 \wedge \Pi_{a,x} \leq \bot_\Pi) \wedge \bigwedge_{v \in V_a \setminus \{\Pi_{a,x} \mid x \in X_a\} \setminus V_{a,delib}} (\mathsf{keep\ stable}\ v) \tag{5.2}$$

The first conjunct expresses that the length of the interval representing the execution of the deliberation is finite. The second conjunct expresses that all priority variables $\Pi_{a,x}$ of the agent a are assigned a non-negative value in the final state of the deliberation interval. The third conjunct denotes that all other *Agent State Variables* maintain their values throughout this interval. The set of *Deliberation Auxiliary Variables* is in this case the empty set ($V_{a,delib} = \emptyset$).

Explicit Definition

The specification of **external** for the deliberation phase has the advantage that reasoning about the agent's behaviour is possible without the concrete knowledge of the implementation of its decision making component. However, it is also desirable to be able to specify the behaviour of this component explicitly as a reaction on the current state of the agent (see for example Listing 5.3). In this case the deliberation section can define additional auxiliary variables and an explicit statement S that defines the behaviour of the agent's priority variables. The behaviour of the deliberation phase is determined by the statement S and has the following semantics:

$$delib_{a,explicit} \;\widehat{=}\; delib_{a,external} \wedge [\![S]\!]_{V_a}$$

The behaviour of the deliberation as described by $delib_{a,external}$ is further restricted by the statement. $[\![S]\!]_{V_a}$ denotes the semantics of the statement S with respect to the agent state variables V_a. The statement semantics is provided subsequently in Section 5.3.3. This means that any statement must assign the priority variables in such a way that they comply with the specification of $delib_{a,external}$.

Although it is possible to restrict the statement semantics for the deliberation section to allow only the assignment of non-negative values to priority variables, only a single statement semantics is used in this work to avoid the additional complexity that arises from two slightly different statement semantics for deliberation and actions. This means that care needs to be taken when defining the deliberation step, as it is possible to violate the specification of $delib_{a,explicit}$ by assigning negative values to priority variables. An

example of this would be the statement `inc:=-1`, as it would assign the priority value $\Pi_{a,inc}$ to a negative value, violating the constraint in the final state of the deliberation.

The language that is used here to describe the reactive deliberation of an agent is very simple, however higher-level language constructs to express the reaction in a more behaviour-oriented fashion can easily be constructed using these primitives. In the following $delib_a$ denotes the behaviour of the agent during its deliberation phase, and is either $delib_{a,external}$ if the deliberation is specified to be *external* or $delib_{a,explicit}$ if an explicit statement is provided that defines the behaviour.

5.3.3 Statement Semantics

The statement semantics defined in this section applies to statements in the deliberation section and to statements that are part of the action definition of an agent. The semantics is inductively defined with respect to a specific set of variables that are controlled by this statement. This technique effectively models the behaviour of *memory variables* using ITL state variables, by maintaining the values of all variables in the set, which are not currently modified by the statement. For the detailed informal description of the statements refer to Section 5.2.2.

Idle

$$[\![\ \boxed{\textsf{idle}}\ ]\!]_V \mathrel{\widehat{=}} \textsf{skip} \wedge \bigwedge_{u \in V} \textsf{stable}\,(u) \tag{5.3}$$

Defines an interval of length one (skip) over which all state variables u in the set V are maintained (stable u). This models the behaviour of memory variables: all variables in the set V maintain their value. Similarly the assignment statements maintain all variables in V that are not assigned by the statement.

Concrete Multiple Assignment

$$[\![\ \boxed{x_0, \ldots, x_n := e_0, \ldots, e_n}\ ]\!]_V \mathrel{\widehat{=}} \textsf{skip} \wedge \bigwedge_{0 \le i \le n} ((\bigcirc x_i) = e_i) \wedge \bigwedge_{u \in V \setminus \{x_0, \ldots, x_n\}} \textsf{stable}\,(u) \tag{5.4}$$

Defines an interval of length one (skip) over which all state variables on the left-hand side of the assignment symbol are assigned in the next state the values that the corresponding expressions on the right-hand side yield in the initial state. Additionally all state variables u in V that are not assigned in the statement maintain their value (stable u).

Temporal Multiple Assignment

$$[\![\ \boxed{x_0, \ldots, x_n \leftarrow\ e_0, \ldots, e_n}\ ]\!]_V \mathrel{\widehat{=}} \bigwedge_{0 \le i \le n} (x_i \leftarrow e_i) \wedge \bigwedge_{u \in V \setminus \{x_0, \ldots, x_n\}} \mathsf{stable}\,(u) \qquad (5.5)$$

Defines an interval of finite length in which all state variables on the left-hand side of the assignment symbol are assigned in the final state of that interval the values that the corresponding expressions on the right hand side yield in the initial state (temporal assignment, $x_i \leftarrow e_i$). Additionally all state variables u in V that are not assigned in the statement maintain their value throughout the interval (stable u).

Sequence

$$[\![\ \boxed{S_1\ ;\ S_2}\ ]\!]_V \mathrel{\widehat{=}} [\![\ \boxed{S_1}\ ]\!]_V\ ;\ [\![\ \boxed{S_2}\ ]\!]_V \qquad (5.6)$$

Defines an interval that can be decomposed into a prefix interval in which the semantics of S_1 holds and a suffix interval in which the semantics of S_2 holds.

Conditional Choice

$$[\![\ \boxed{\textsf{if } w \textsf{ then } S_1 \textsf{ else } S_2}\ ]\!]_V \mathrel{\widehat{=}} (w \wedge [\![\ \boxed{S_1}\ ]\!]_V) \vee (\neg w \wedge [\![\ \boxed{S_2}\ ]\!]_V) \qquad (5.7)$$

$$[\![\ \boxed{\textsf{if } w \textsf{ then } S_1}\ ]\!]_V \mathrel{\widehat{=}} (w \wedge [\![\ \boxed{S_1}\ ]\!]_V) \vee (\neg w \wedge \mathsf{empty}) \qquad (5.8)$$

Equation (5.7) defines the general case; Equation (5.8) the special case that the *else* part is omitted. If the state formula w holds in the initial state then the interval is defined by the semantics of S_1, otherwise it is defined by the semantics of S_2. For the special case: if the state-formula w does not hold in the initial state, the interval is a single state (length zero: empty).

Iteration

$$[\![\ \boxed{\textsf{for } K < e \textsf{ do } S}\ ]\!]_V \mathrel{\widehat{=}} \exists K, n \cdot n = \max(e, 0) \wedge [\![\ \boxed{K \leftarrow\ 0}\ ]\!]_{V \cup \{K\}}\ ;\ ([\![\ \boxed{S\ ;\ K := K + 1}\ ]\!]_{V \cup \{K\}})^n \qquad (5.9)$$

Defines a finite interval in which the maximum number of iterations (n) is evaluated in the initial state of the interval. The *max* function ensures that this value is non-negative. The interval is decomposed into a prefix and a suffix interval. In the prefix interval the loop variable K is allocated and initialised to the value 0. For this the temporal assignment statement ($\boxed{K \leftarrow\ 0}$) is used to allow for concrete implementations that take more than one unit-interval. In the suffix interval the iteration of the statement S and the increment

of the counter are captured $((\ldots)^n)$. It is decomposed into n subintervals, for which the semantics of S holds over a prefix and the concrete assignment, that increments the loop variable, holds over a suffix. Note that the value of the loop variable K is maintained during the execution of the statement, as the semantics of the statement is defined with respect to the augmented set $V \cup \{K\}$.

Local Variables

$$[\![\boxed{\textsf{var } X_0, \ldots X_n \leftarrow\ e_0, \ldots, e_n : S}]\!]_V \ \widehat{=}\ \exists X_0, \ldots, X_n \cdot \quad (5.10)$$
$$[\![\boxed{X_0, \ldots X_n \leftarrow\ e_0, \ldots, e_n : S}]\!]_{V \cup \{X_0, \ldots, X_n\}}$$

Introduces a set of local variables. As for the iteration variable before, the temporal multiple assignment is used to allow for concrete implementations to define the time that is required for the allocation and initialisation of the variables. The values of the local variables are maintained, unless they are explicitly modified in the statement.

Terminate The termination statement **terminate** can be executed by an agent. It is an abbreviation to set the control variable $doterm_a$ of the agent a to *true*.

$$[\![\boxed{\textsf{terminate}}]\!]_V \ \widehat{=}\ [\![doterm_a \leftarrow\ true]\!]_V \quad (5.11)$$

The terminate statement can be used in any statement and assigns the variable $doterm_a$ of the agent executing the statement to *true*. At the end of the next execution phase, the agent will break the deliberation-enforcement-execution cycle and enter the termination phase.

5.3.4 Enforcement Phase

After the deliberation phase the agent enters the enforcement phase of its execution cycle. This phase implements vigilant mechanisms for the enforcement of security policies on the agent. The detailed description of this phase is delayed to Chapter 8, and only a high level specification is provided at this point.

The specification is similar to the external specification of the deliberation phase, however, the priority variables $\Pi_{a,x}$ can be assigned to negative values. With the exception of the value 0 lower values denote a higher priority, allowing for the enforcement of *obligations* by assigning them a negative priority value. This guarantees that any obligation enforced by vigilant mechanisms can overrule the preferences decided by the deliberation phase. For implementation concerns an upper boundary $\top_\Pi$ is introduced, to denote the

highest priority.

$$enf_a \mathrel{\widehat{=}} \left(\mathsf{finite} \wedge \bigwedge_{x \in X_a} \mathsf{fin}\,(\top_\Pi \leq \Pi_{a,x} \leq \bot_\Pi) \wedge \bigwedge_{v \in V} (\mathsf{stable}\ v) \right) ; [\![\boxed{ready_a := true}]\!]_{V_a} \tag{5.12}$$

Where $V = V_a \setminus \{\Pi_{a,x} |\ x \in X_a\} \setminus V_{a,enf}$, the set of all agent state variables, except the priority and enforcement auxiliary variables. Equation (5.12) states that the enforcement phase is decomposed into a finite prefix interval and a suffix interval of length one. In the prefix the agent state variables, except the priority variables $\Pi_{a,x}$ and auxiliary enforcement variables $V_{a,enf}$, keep their values. All priority variables assume in the last state a value that is between the lower and upper bound of priority values $(\top_\Pi, \bot_\Pi)$.

The suffix indicates that the agent is *ready* for the execution of an action by setting the control variable $ready_a$ to *true*. Subsequently the agent enters its *execution phase*.

5.3.5 Execution Phase

In the execution phase the action that has been assigned the highest priority is chosen for execution, provided its precondition is *true*. This is captured in the *functional guard* $g_{a,x}$ of the action x.

Functional Guard The functional guard $g_{a,x}$ of an agent a's action x is *true* iff the corresponding precondition $p_{a,x}$ is true and the action has the highest priority $\Pi_{a,x}$ amongst all those actions for which the precondition is fulfilled.

$$g_{a,x} \mathrel{\widehat{=}} p_{a,x} \wedge \Pi_{a,x} \neq 0 \wedge \bigwedge_{x' \in X_a} (\Pi_{a,x} \leq \Pi_{a,x'} \vee \neg p_{a,x'} \vee \Pi_{a,x'} = 0) \tag{5.13}$$

The mechanism is best explained using a small example. Let an agent a define three actions x_0, x_1 and x_2, with the preconditions of these actions being $p_{a,x_0} = false$, $p_{a,x_1} = true$ and $p_{a,x_2} = true$. Assuming the agent prioritises its actions as follows: $\Pi_{a,x_0} = 1$ (highest priority, assignable in the *deliberation phase*), $\Pi_{a,x_1} = 0$ (must not execute) and $\Pi_{a,x_2} = \bot_\Pi$ (lowest priority) then the guards will evaluate as follows:

$$
\begin{array}{lll}
g_{a,x_0} & = false & \text{as a direct consequence of } p_{a,x_0} = false \\
g_{a,x_1} & = false & \text{as a direct consequence of } \Pi_{a,x_0} = 0 \\
g_{a,x_2} & = f_{x_2x_0} \wedge f_{x_2x_1} \wedge f_{x_2x_2} & \text{where } f_{xx'} = (\Pi_{a,x} \leq \Pi_{a,x'} \vee \neg p_{a,x'} \vee \Pi_{a,x'} = 0) \\
 & = true & \text{because} \\
f_{x_2x_0} & = true & \text{as a direct consequence of } p_{a,x_0} = false \\
f_{x_2x_1} & = true & \text{as a direct consequence of } \Pi_{a,x_1} = 0 \\
f_{x_2x_2} & = true & \text{as a direct consequence of } \Pi_{a,x_2} = \Pi_{a,x_2}
\end{array}
$$

It follows that only the guard g_{a,x_2} is true and therefore the agent can only choose x_2 for execution. The intuition is that the priority can never overrule the precondition and only those actions with the highest priority (different from zero) can have a *true* functional guard.

Given the intuition that the action with the smallest priority value different from zero has a *true* functional guard, the assignment of a negative priority value in the enforcement phase could overrule any priority decision made in the deliberation phase. This way the obligations that are specified in vigilantly enforced policies can be enforced. An action x for which the functional guard $g_{a,x}$ is *true* is said to be *enabled* if the agent is *ready* to execute.

Enabled Local Action An action $x \in X_a$ is enabled if the agent a is *ready* and the functional guard $g_{a,x}$ of the action is *true*.

$$enabled_{a,x} \mathrel{\widehat{=}} g_{a,x} \wedge ready_a = true \tag{5.14}$$

In this chapter we consider only the execution of local actions that involve the agent itself in the execution. Chapter 6 extends this definition to synchronise the agent and the object for *remote actions*. The control variable $ready_a$ is set to true at the end of the agent's enforcement phase (see Section 5.3.4, Equation (5.12)).

Local Action

An action can only be executed if it is *enabled*. When executed, the semantics of the action indicates that the agent started the execution by setting the control variable $ready_a$ to *false*, viz. no other action can be enabled. Subsequently the statement $S_{a,x}$ of the action is executed. This is denoted by $stat_{a,x}$ (defined subsequently in Equation (5.16)). The

semantics of local actions is denoted by $\psi_{a,x}$, defined in Equation (5.15).

$$\psi_{a,x} \mathrel{\widehat{=}} enabled_{a,x} \wedge [\![\boxed{ready_a := false}]\!]_{V_a} \; ; stat_{a,x} \tag{5.15}$$

The conjunct $enabled_{a,x}$ guarantees that the agent is ready for execution and that the functional guard $g_{a,x}$ of the action is *true*. The control variable $ready_a$ is set to false in the second state of the execution, guaranteeing that $ready_a$ is *true* only in the initial state. The semantics of the statement execution is denoted by $stat_{a,x}$.

Statement Execution Access Control and Obligation decisions for the agent have already been enforced in the *enforcement phase* of the agent's execution cycle. Integrity policies however cannot be enforced beforehand and require evaluation *after* the effect of the statement execution is known. Consequently the semantics of an action must allow for the failure of the action due to violated integrity constraints. This is captured in the semantics of the statement execution. Recall from Section 3.2 that the failure of an action means that all agent variables keep the values they had in the initial state of the execution. This semantics is captured by performing the computation on a set of local variables $v'_0, \ldots, v'_j$ of the agent variables $v_0, \ldots, v_j$ and copying their values after a successful integrity check back to the agent variables (denoted by $succeed_{a,x}$, defined in Equation (5.17)). If any integrity constraints are violated the values are simply discarded (denoted by $fail_{a,x}$, defined in Equation (5.18)). This is expressed in the following Equation (5.16).

$$stat_{a,x} \mathrel{\widehat{=}} \exists v'_0, \ldots, v'_j \cdot ([\![\boxed{v'_0, \ldots v'_j \leftarrow v_0, \ldots, v_j}]\!]_{V_a^+} ; [\![S'_x]\!]_{V_a^+} ; (succeed_{a,x} \oplus fail_{a,x})) \tag{5.16}$$

V_a^+ denotes the here the set $V_a \cup \{v'_0, \ldots, v'_j\}$, that is the set of agent state variables augmented by the local copies of agent variables that are assigned in the statement $S_{a,x}$. The set of assigned variables can be obtained by static analysis at compilation time. The statement $S'_{a,x} = S_{a,x}[v'_0, \ldots, v'_j / v_0, \ldots, v_j]$ denotes the statement $S_{a,x}$ where every occurrence of a variable $v \in \{v_0, \ldots, v_j\}$ is replaced by the corresponding local variable $v' \in \{v'_0, \ldots, v'_j\}$. This guarantees that the agent variables $V_{a,decl}$ are not changed during the execution of $S'_{a,x}$. The statement semantics is the same as defined in Section 5.3.3. The operator $\oplus$ denotes non-deterministic choice, viz. either $succeed_{a,x}$ or $fail_{a,x}$ are true, but not both. The choice is left non-deterministic, because the failure can only occur due to security decisions, that are determined by policies. The non-deterministic choice is refined into a deterministic choice with the implementation of enforcement mechanisms (see Chapter 9).

Action Succeeds The successful execution of the action is described by $succeed_{a,x}$. The successful execution of the action x is indicated by setting the control variable $done_{a,x}$.

$$succeed_{a,x} \mathrel{\widehat{=}} [\![\boxed{done_{a,x}, \overline{done_{a,x}}, v_0, \ldots, v_j \leftarrow \texttt{true}, \overline{false}, v'_0, \ldots, v'_j}]\!]_{V_a^+} \tag{5.17}$$

We denote here by $\overline{done_{a,x}}$ the complementary list of all control variables

$$\overline{done_{a,x}} \mathrel{\widehat{=}} \{done_{a,x'} \mid x' \in X_a \setminus \{x\}\} \cup \{failed_{a,x'} \mid x' \in X_a\}$$

and by $\overline{false}$ a list of *false* literals that has the same length as $\overline{done_{a,x}}$. This means that all control variables that indicate the failure of an action for this agent are set to *false*, as are all control variables indicating success but the variable $done_{a,x}$.

Although this may seem to be a large overhead for the maintenance of the control variables, a concrete implementation can take advantage of the fact that only one of the control variables can be true at the end of the action execution (per agent). For example, encoding the Boolean control variables in a single signed integer variable allows to implement the setting of the control variables as a single (multiple) assignment. The encoding would enumerate the actions contained in the agent and indicate by a positive value the successful execution of the corresponding action and with a negative value its failure. A value of zero would denote that none of the variables is *true*, indicating that none of the actions has been executed in the last cycle.

Action Fails The failure of the execution is described by $fail_{a,x}$. The failed execution of the action x is indicated by setting the control variable $failed_{a,x}$.

$$fail_{a,x} \mathrel{\widehat{=}} [\![\boxed{failed_{a,x}, \overline{failed_{a,x}} \leftarrow \texttt{true}, \overline{false}}]\!]_{V_a^+} \tag{5.18}$$

Similarly to the successful execution the control variables are set to indicate failure. Here $\overline{failed_{a,x}}$ denotes the complementary list of control variables.

$$\overline{failed_{a,x}} \mathrel{\widehat{=}} \{failed_{a,x'} \mid x' \in X_a \setminus \{x\}\} \cup \{done_{a,x'} \mid x' \in X_a\}$$

It is not necessary to define the values of the agent variables in the case of failure, as their value remained unchanged during the execution of the rewritten statement $S'_{a,x}$.

Choice of Execution In the execution phase one of the agent's enabled actions is chosen for execution non-deterministically. In the case that none of the agent's *local* actions is enabled, the agent remains *idle*, viz. it continues with the execution of the deliberation phase. The fact that the agent remained *idle* is indicated by the control variables: none of

the agent's control variables $done_{a,x}$ or $failed_{a,x}$ is *true*. This is captured by the definition of the execution phase in Equation (5.19).

$$exec_a \mathrel{\widehat{=}} ((\bigoplus_{x \in X_a} \psi_{a,x}) \oplus idle_a) \,;\, [\![\textbf{if } doterm_a \textbf{ then } term_a := true]\!]_{V_a} \tag{5.19}$$

Exactly one of the agent's actions is chosen for execution. Since the semantics of the action $\psi_{a,x}$ demands that the action is enabled (see Equation (5.15)) the operator exclusive-or represents a non-deterministic choice between the enabled actions, only. Alternatively the agent may remain *idle* (denoted by $idle_a$). The conditional choice at the end of the execution phase ensures that the control variable $term_a$, indicating the termination condition for the agent a is set to *true*. The use of two variables to capture the termination of the agent are necessary: $doterm_a$ indicates that the agent wants to terminate at the end of the next deliberation-enforcement-execution cycle, whereas the control variable $term_a$ marks the final state of this cycle. This becomes clear in Section 5.3.7, where the criteria for the agent to leave the deliberation-enforcement-execution cycle and enter the termination phase is defined.

Remaining Idle The possibility to remain *idle* originates from the fact that there may not be an *enabled* action. This intuitively would mean that *idle* describes the behaviour of the agent only if none of the actions is enabled. This view has been taken in [124]. However, it can be shown that there are cases where the choice to remain *idle* is required to satisfy the fairness criterion that is discussed later in Section 5.3.7. Consequently the choice to remain *idle* is here defined as an unconditional alternative to the execution of an action. Equation (5.20) defines the semantics of *idle*.

$$idle_a \mathrel{\widehat{=}} [\![\boxed{ready_a := false \,;\, \overline{done_a} \leftarrow \overline{false}}]\!]_{V_a} \tag{5.20}$$

Here $\overline{done_a}$ denotes the list of all *agent control variables* $done_{a,x}$ and $failed_{a,x}$ of the agent a; $\overline{false}$ denotes the corresponding set of *false* literals.

5.3.6 Termination Phase

The termination phase of an agent means that the agent will leave its deliberation-enforcement-execution cycle and cease to execute any actions. The agent stutters until the distributed termination of the system. This is defined in Equation (5.21).

$$terminate_a \mathrel{\widehat{=}} ([\![\textbf{idle}]\!]_{V_a})^* \tag{5.21}$$

The semantics of the **idle** statement specifies that the control variable $ready_a$ remains *false*, ensuring that the agent cannot participate in the execution of an action.

5.3.7 Agent Semantics

The semantics of a single agent captures the initialisation of the agent (Section 5.3.1) and the iteration of its *deliberation* (Section 5.3.2), *enforcement* (Section 5.3.4) and *execution* (Section 5.3.5) phases. The *fairness* property (denoted by $fair_a$, subsequently defined in Equation (5.23)) places an additional constraint on the non-deterministic choice of actions that is described in Equation (5.19).

$$\varphi_a \mathrel{\widehat{=}} init_a \,;\, ((delib_a \,;\, enf_a \,;\, exec_a)^* \wedge fair_a \wedge \mathsf{halt}\,(term_a = true)) \,;\, terminate_a \tag{5.22}$$

Termination Criteria The agent terminates the iteration of its DEE cycle iff the value of $term_a$ is *true*. The definition of the halt statement uses an equivalence, that makes it necessary to use two variables to capture the termination. The variable $doterm_a$ can be *true* during the deliberation, enforcement, execution phase, *without* forcing the termination. The definition in Equation (5.19) ensures that the variable $term_a$ can only be *true* in the final state of the execution phase, that also represents the final state of the cycle, as defined by the halt statement.

Fairness The fairness property is described analogous to that in [124].

$$fair_a \mathrel{\widehat{=}} \mathsf{inf} \supset \Box \left(\bigwedge_{x \in X_a} ((\Box\Diamond enabled_x) \supset \Diamond \psi_x) \right) \tag{5.23}$$

Equation (5.23) states that given an infinite execution, any action that is infinitely often enabled is also executed infinitely often. This property prevents the total *starvation* of one action. Fairness on a finite execution interval is a scheduling problem and not further addressed in this work.

5.4 Multi-Agent System Semantics

The semantics of the MAS is the concurrent execution of all agents that terminates distributively when the termination condition of all agents hold simultaneously (sufficient termination condition). This is expressed in Equation (5.24)

$$SMAS_1 \mathrel{\widehat{=}} \left(\mathsf{halt}\,(\bigwedge_{a \in \mathcal{A}} term_a = true) \wedge \bigwedge_{a \in \mathcal{A}} \varphi_a \right) \tag{5.24}$$

We assume here that the global system state is implicitly composed of all the agents' states. This can be safely assumed, as all agent state variables are not shared between agents. Communication between agents is modelled using the concept of objects that is detailed in Chapter 6.

The following section illustrates the semantics of the $SMAS_1$ using the example provided in Listing 5.3 on Page 112.

5.5 Example of $SMAS_1$

This section discusses the example that has been used to motivate the informal semantics of an $SMAS_1$ in Section 5.2.4 with respect to its formal semantics described in Section 5.3. The listing is here included again for the readers convenience.

Listing 5.4: Example Agent Specification

```
1  /* Agent Definition */
2  agent ag :
3
4      /* Agent Variables */
5      var x = 0
6
7      /* Agent Actions */
8      when x < 2^32 - 1 do inc : x := x+1
9      when x < 2^31 do dbl : x := 2*x
10
11     /* Deliberation */
12     deliberation : {
13     if x % 2 = 0 then inc,dbl ← 1,0
14                  else inc,dbl ← 0,1 }
15 end
```

The following figure shows the behaviour of the $SMAS_1$ system consisting of only the agent ag.

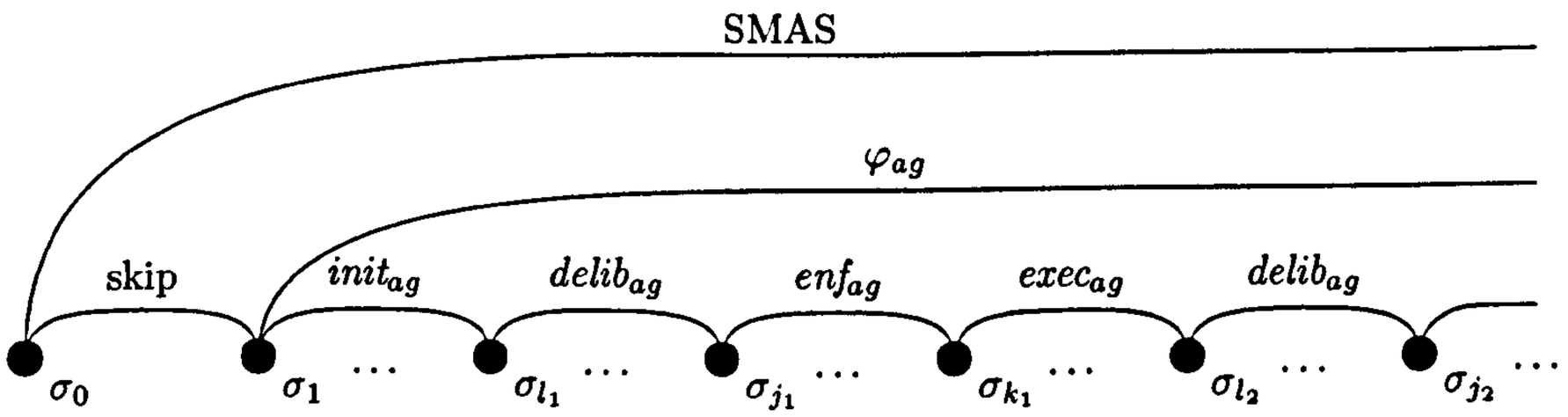


Figure 5.1: Specification level behaviour of the $SMAS_1$ example

The SMAS is undefined in the initial state (σ_0). This is required for the composition of

SMAS systems. The agent execution starts with its *initialisation phase*. The initialisation $init_{ag}$, defined in Equation (5.1), takes finite time and is completed in state σ_{l_1}. It follows the *deliberation phase* $delib_{ag}$ of the agent, defined in Equation (5.2). Again the execution takes finite time, and terminates in state σ_{j_1}. After the deliberation the *enforcement phase* enf_{ag}, defined in Equation (5.12), is executed ending in state σ_{k_1}. The agent then enters the *execution phase*, defined in Equation (5.19). Here the functional guards are evaluated and one of the enabled actions is chosen for execution non-deterministically. After the execution phase the agent has completed the first iteration of its *deliberation-enforcement-execution* cycle.

Initialisation Phase The initialisation phase assigns the initial values to all agent state variables. The agent state variables are in this example:

$$V_{ag} = \{ready_{\mathtt{ag}}, doterm_{\mathtt{ag}}, term_{\mathtt{ag}}, done_{\mathtt{ag,inc}}, done_{\mathtt{ag,dbl}},$$
$$failed_{\mathtt{ag,inc}}, failed_{\mathtt{ag,dbl}}, \Pi_{\mathtt{ag,inc}}, \Pi_{\mathtt{ag,dbl}}, \mathtt{x}\}$$

All variables are assigned their default values at the end of the initialisation phase, in state σ_{l_1}, viz. the priority variables $\Pi_{a,x}$ are initialised to 1, the agent variable x is set to its defined initial value 0, and the other control variables to *false*. The variable $ready_{\mathtt{ag}}$ is *false throughout* the initialisation phase.

Deliberation Phase The deliberation phase in the example checks whether the current value of x is even or odd. If it is even, the highest assignable preference value ($\Pi_{\mathtt{ag,inc}} = 1$) is assigned to the action `inc` and the preference value ($\Pi_{\mathtt{ag,dbl}} = 0$) indicates that action `dbl` must not be executed. If x is odd, the priority assignments are reversed.

Since the initialisation phase did assign the initial value 0 to x, the priority values in the first iteration of the deliberation-enforcement-execution cycle will be $\Pi_{\mathtt{ag,inc}} = 1$ and $\Pi_{\mathtt{ag,dbl}} = 0$. In either case the priority variables are set using the *temporal assignment*, guaranteeing that the deliberation terminates in finite time (viz. state σ_{j_1}).

Enforcement Phase The SMAS$_1$ system specification is not concerned with the enforcement of policies. We therefore assume that the high-level specification provided in Equation (5.12) does not change any of the priority variables, viz. no policy is implemented. The enforcement phase is finite and terminates in state σ_{k_1}.

Execution Phase In this phase all the functional guards are evaluated. The preconditions of both actions hold. Given the priority assignment it follows that the only enabled

action is `inc` (see Equations (5.13) and (5.14)). The behaviour of the execution of action `inc` is depicted in Figure 5.2:

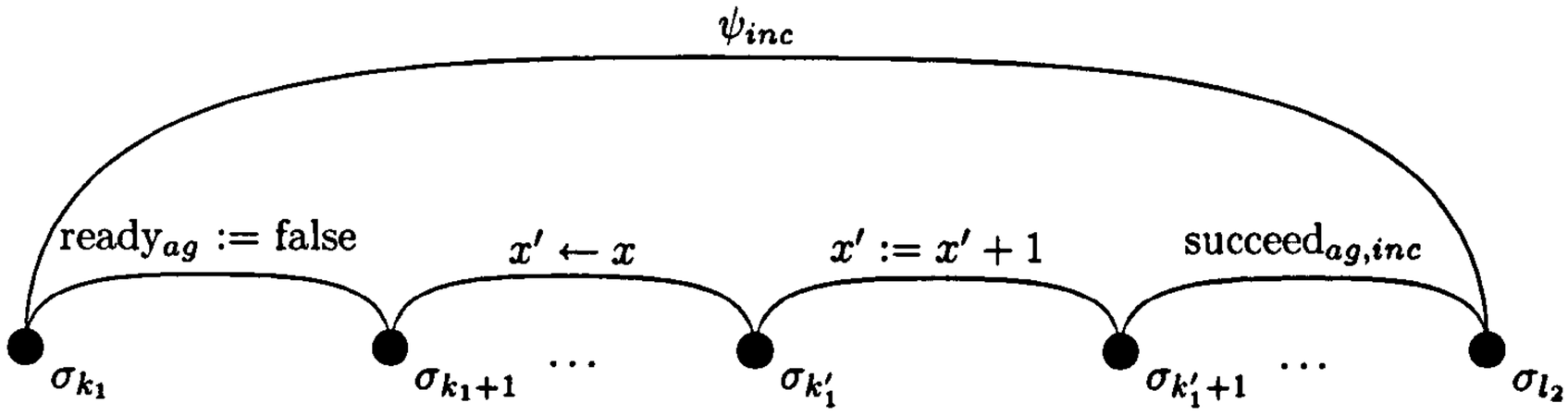


Figure 5.2: Specification level behaviour of action `inc`

Initially the agent is locked by setting the control variable $ready_{\mathtt{ag}}$ to *false*, viz. the agent cannot execute any other action. The setting of the variable is defined to take exactly one time step. Subsequently the agent's state is copied into the local variables. The specification does define that this happens in finite time and that in the final state all local variables (here x') have the same values that the corresponding agent variables (here `x`) had in the initial state of the assignment. The final state of the assignment is denoted by $\sigma_{k'_1}$.

The statement is executed on the *local* variables. As the statement uses the concrete assignment we know that this takes exactly one time step. If the action succeeds, the now updated values of the local variables are subsequently copied back into the agent variables, representing that the action does actually take effect and the control variable $done_{\mathtt{ag,inc}}$ is set to *true* (defined in $succeed_{\mathtt{ag,inc}}$, Equation (5.17)). Alternatively the specification allows the action to fail, i.e. the agent variables remain unchanged and the control variable $failed_{\mathtt{ag,inc}}$ is set to *true* (defined in $fail_{\mathtt{ag,inc}}$, Equation (5.18)). Both possibilities take finite time and terminate in state σ_{l_2}, in which the action execution terminates.

This completes the first deliberation-enforcement-execution cycle of the agent. The agent will subsequently reenter its deliberation phase, where the assignment of the priority variables will be reversed. Table 5.2 lists the values of the agent and control variables in the described states. The "?" indicates, that the value of the variable is not defined in the state.

The specification uses the temporal assignment. A concrete implementation must refine this assignment into either a concrete parallel assignment or a sequence of simple assignments. The refinement of this example is provided in Chapter 9, Section 9.3.

Agent Variables		σ_0	σ_1	σ_{l_1}	σ_{j_1}	σ_{k_1}	σ_{l_2}	σ_{j_2}
x	Variable x	?	?	0	0	0	1	1
	Priority Variables							
$\Pi_{ag,inc}$	Priority of `inc`	?	?	1	1	1	1	0
$\Pi_{ag,dbl}$	Priority of `dbl`	?	?	1	0	0	0	1
	Control Variables							
$ready_{\mathtt{ag}}$	ag is not *ready*	?	F	F	F	T	F	F
$done_{\mathtt{ag,inc}}$	ag successfully executed `inc`	?	?	F	F	F	T	T
$failed_{\mathtt{ag,inc}}$	ag failed executing `inc`	?	?	F	F	F	F	F
$done_{\mathtt{ag,dbl}}$	ag successfully executed `dbl`	?	?	F	F	F	F	F
$failed_{\mathtt{ag,dbl}}$	ag failed executing `dbl`	?	?	F	F	F	F	F

Table 5.2: States in the Execution of Listing 5.3

5.6 Summary

This chapter defined the syntax and the specification oriented semantics of the $SMAS_1$ subset of the SANTA language. The subset allows for the specification of fully autonomous agents, that are executing local actions to perform computations on their encapsulated variables. The execution of an agent is staged in phases. The execution starts with the *initialisation phase*, where the agent state is initialised (Section 5.3.1). Subsequently the agent enters its *deliberation-enforcement-execution* cycle. These three phases are constantly iterated until the agent decides to terminate.

In the *deliberation phase* (Section 5.3.2) the agent prioritises its actions for execution. In the *enforcement phase* (Section 5.3.4) these priorities are reassigned to ensure the compliance with vigilantly enforced policies. In the execution phase (Section 5.3.5) the action with the highest priority is executed, provided its *precondition* is *true*. The agent can choose to terminate by executing the **terminate** statement, and leave the cycle to enter the *termination phase*. In the termination phase the agent remains **idle**, allowing for the distributed termination of the system.

The advantage of the separation of the phases is the increased modularity of the system description. The deliberation phase can be used to encode (intelligent) decision making algorithms. These may be specified explicitly or left undefined using the keyword **external** indicating that their implementation is not formalised within the framework. The notion of priority variables provides a clear interface to the rest of the system. The enforcement phase encodes enforcement mechanisms for policies; this is detailed later in Chapter 8. Finally the execution phase captures the behaviour of the actual action execution. The execution either *succeeds*, viz. the agent variables are updated, or *fails*, viz. the agent variables remain unchanged. This is indicated in the control variables.

An example of a single agent that is alternating between the execution of two action has been described at the specification level, where only the states between the phases in

the execution are considered. The behaviour of all agent state variables over these states has been detailed.

Not addressed in the $SMAS_1$ subset is the existence of a shared environment through which agents can communicate and the existence of policies and their various enforcement mechanisms. The shared environment is introduced using the notion of *objects* in the next chapter as the $SMAS_2$ subset of the SANTA design language. Chapter 7 introduces the syntax and specification oriented semantics of policies that are then linked with the behaviour of the system by enforcement mechanisms in Chapter 8.

Chapter 6

Objects

Objects represent the shared environment of agents. The focus is on the formalisation of objects as passive entities and the communication between agents and objects using remote actions. Key is the formalisation of different points of failure in the communication due to policy decisions.

6.1 Introduction

Having introduced SANTA agents and their semantics we augment the $SMAS_1$ with the notion of objects. Objects are passive entities that represent the agents' shared environment. Agents can invoke object interfaces and thus modify or perceive the state of their environment. The invocation of object interfaces is synchronous. This chapter describes how objects are specified in the $SMAS_2$ subset of the language and how agents can use objects as a means of sensing and affecting their environment.

The emphasis is on the execution of an agent's *remote action* and the various points in which a remote action can fail due to policy decisions. As in the preceding chapter, the actual condition for this failure remains undefined and captures only the effect that a policy decision has on the result of the communication.

This chapter is structured as follows. In Section 6.2 the additional syntactic constructs are defined and described informally. In Section 6.3 the semantics of a single agent is extended to include the execution of remote actions. Following this, the behaviour of objects is formalised in Section 6.4. These definitions are combined in Section 6.5 to yield the overall semantics of the $SMAS_2$ subset of the SANTA language. The chapter concludes with an example of an $SMAS_2$ program in Section 6.6 and a brief summary in Section 6.7.

6.2 Objects in SANTA

The following syntactic constructs are introduced to describe objects. Only the new or augmented production rules are included in the Listing 6.1

Listing 6.1: EBNF for $SMAS_2$

```
1  MAS       = {agent | object}.
2
3  object    = 'object', id, ':', { vardekl | intdekl }, 'end'.
4
5  actdekl   = 'when', expr, 'do', id, ':', (statement | external).
6
7  external = [ lit ':' ] id, '.', id, '(', [ vallist ], ')' .
8
9  intdekl   = id, '(', [ paramlist ], ')', ':', statement, '.' .
10
11 paramlist= ['in'], ['out'], id, { ',', ['in'], ['out'], id }
12
13 vallist   = (  expr, { ',', expr } ) .
```

The $SMAS_2$ system specification contains additionally to the set of agents also a set of objects $\mathcal{O}$ that constitute the agents' shared environment. An object specification is introduced by the keyword **object** followed by its system wide unique identifier `id`. Similar

to an agent definition, an object definition encapsulates a set of variables. Instead of actions an object provides a set of interfaces that are accessible by other entities in the system. The definition of an object closes with the keyword **end**.

6.2.1 Object State Variables

Object variables are defined in the same way as agent variables. They denote the explicitly declared state of the object, that can be only modified by the execution of interfaces. The set of object variables is denoted by $V_{o,decl}$. Additionally, auxiliary variables are part of the object's state. These are control variables, denoted by the set $V_{o,contr}$, and enforcement variables, denoted by the set $V_{o,enf}$.

The set of control variables contains the variable $ready_o$ that indicates that the object is accessible by an agent. The control variables $done_{o,a,i}$ and $failed_{o,a,i}$ indicate the success or failure of the previous invocation of interface i by agent a. Contrary to the control variables for agents they are indexed with three subscripts. This allows to reference not only the past execution of interfaces in policies but also the agents that invoked these interfaces. The set of enforcement variables accommodates all variables that are required by the vigilant enforcement mechanism to enforce the object's policies.

The set of all *object state variables*, denoted by V_o, is the union:

$$V_o \mathrel{\widehat{=}} V_{o,decl} \cup V_{o,enf} \cup V_{o,contr}$$

In the semantics all variables in the set V_o are modelled as ITL state variables. Similar to the case discussed in the previous chapter these are explicitly controlled by the statement semantics and resemble the behaviour of *memory variables*.

6.2.2 Object Interfaces

An object interface is introduced by an object wide unique identifier followed by an optional parameter list that is enclosed in parenthesis. The list defines parameters that are designated as input parameters by the keyword **in** or as output parameters by the keyword **out**. If a designator is omitted the parameter is assumed to be used for both, input and output. Parameters are passed by value. The statement describing the behaviour of the interface follows the colon. The semantics of the statements is the same as previously introduced in Section 5.3.3, however, the use of the keyword **terminate** is prohibited.

6.2.3 Remote Actions

The syntax for action definitions has been augmented to allow for the execution of remote actions. The effect that the action has on the environment is defined by the invoked

object interface, which is defined in the remote action by the object's identifier and the interface identifier. The optional parameter list allows for the exchange of information between the agent's state and the state of the invoked object. If the parameter is defined in the interface as an output parameter the expression is syntactically restricted to be the identifier of an agent variable.

Time Out As the concrete implementation of the object interface is generally unknown to the agent, a time-out for the execution of external actions is provided. The time-out is optional and precedes the object name in the invocation. If the computation of the interface exceeds the specified timeout, the system can abort the computation and both the agent's and the object's state remain unchanged. The specification does not guarantee that the time the agent is involved in the invocation has a strict upper bound of the specified timeout. However, it guarantees that if the time limit is exceeded before the results of the computation are available, then the action is considered to be failed, and the available results are discarded. A concrete implementation will potentially be able to give guarantees on the worst-case scenario.

Parameterised Interfaces The communication between agent and object is semantically similar to the communication using joint actions as it is defined in [124]. However, interfaces and parameter lists allow to clearly define the data that can be accessed by the calling and the called side without requiring knowledge of the internal structure of the interface. This makes it possible to analyse potential information flow in the system based solely on the definition of object interfaces. This is important as knowledge of the internal structure cannot be assumed in an open environment.

Explicit Naming of Remote Actions The rationale to name the external action within the agent is to be able to control different actions of the agent using policies. An agent may define two external actions that invoke the same object interface albeit with different parameter lists, e.g. to store the results in different output variables. From the agent's point of view both actions are distinct. However, from the viewpoint of the invoked object both are indistinguishable. To be able to define accurate policies that control the behaviour of the agent it must be possible to distinguish between the two different invocations of the same object interface.

This is best explained by the following example. Assume the following specification:

Listing 6.2: Example for explicitly named remote actions

```
1  /* Agent Definition */
2  agent ag :
3      var x = 0
4      var y = 0
5
6      /* Save to variable x */
7      when true do readX : ob.read(x)
8
9      /* Save to variable y */
10     when true do readY : ob.read(y)
11
12     /* ... */
13 end
```

In Listing 6.2 agent `ag` could choose to alternate between executing `readX` and `readY` to keep always the previously read value. Although both actions invoke the same object interface their effect on the agent is different and a policy could make different decisions for each of the actions.

6.2.4 Motivating Example

The example given in this section modifies the example provided in Section 5.2.4. The new example specification is as follows:

Listing 6.3: Motivating Example for $SMAS_2$

```
1  /* Agent Definition */
2  agent ag :
3      /* Agent Variables */
4      var x = 0
5      var y = 0
6
7      /* Agent Actions */
8      when x < 2^32 - 1 do inc : ob.set(x+1, x)
9      when x < 2^31 do dbl : ob.set(2*x, x)
10
11     /* Deliberation */
12     deliberation : {
13     if x % 2 = 0 then inc,dbl ← 1,0 else inc,dbl ← 0,1 }
14 end
15
16 /* Object Definition */
17 object ob :
18     /* Object Variables */
19     var x = 0
20
21     /* Interfaces */
22     set (in a, out b) : {  x,b ← a,a  }
23 end
```

The example in Listing 6.3 introduces the object `ob` that encapsulates the variable `x`. The object provides the interface **set(in a, out b)** that will update the value of the object's variable `x` to the input parameter `a` and returns this value also as the output parameter `b`. The actions `inc` and `dbl` are changed to remote actions that invoke the interface **set(in a, out b)** of the object `ob` with the appropriate new value.

6.3 Single Agent Semantics

This section describes the conservative extensions to the definitions in Section 5.3 that are required to take the execution of remote actions into account. By *conservative* we mean that if none of the constructs that are introduced in the $SMAS_2$ subset are used, then the semantics of the $SMAS_2$ system is equivalent to the semantics of the $SMAS_1$ system.

In the following the set of actions X_a of the agent a is the union of the set of local actions, denoted by X_a^{loc}, and the set of remote actions, denoted by X_a^{rmt}. The sets X_a^{loc} and X_a^{rmt} are disjoint.

$$X_a = X_a^{loc} \cup X_a^{rmt}$$

The *initialisation phase* of the agent remains unchanged. However, the set X_a now additionally contains the set of remote actions. Similarly, the definition of the *deliberation phase* remains unaffected, as does the definition of the *enforcement phase*. In the *execution phase* changes are required to allow for the execution of *remote actions*.

6.3.1 Execution Phase

The definition of the functional guard provided in Section 5.3.5, i.e. Equation (5.13) is retained, however using the enlarged action set X_a. As remote actions must synchronise with their target object the definition of an *enabled action* requires modification.

Enabled Action

Equation (6.1) augments the definition of $enabled_{a,x}$ that was provided in Equation (5.14) to include the readiness of the invoked object o.

$$enabled_{a,x} \mathrel{\widehat{=}} g_{a,x} \wedge \bigwedge_{j \in E_{a,x}} ready_j \tag{6.1}$$

where $E_{a,x}$ denotes the set of system entities that must synchronise for the execution of the action x. The set $E_{a,x}$ is defined as follows:

$$E_{a,x} = \begin{cases} \{a\} & \text{if } x \in X_a^{loc} \\ \{a, o\} & \text{if } x \in X_a^{rmt} \text{ and } x \text{ invokes an interface of object } o \end{cases} \tag{6.2}$$

This is clearly a conservative extension of the original definition because $X_a = X_a^{loc}$ means that $E_{a,x} = \{a\}$ for any $x \in X_a$, rendering Equation (6.1) identical to Equation (5.14). The choice to introduce the set $E_{a,x}$ is based on the fact that the introduction of a *Security Enforcer* later in Chapter 8 will require an additional entity to synchronise with the execution.

Action Semantics

The semantics of the action must be augmented to take into account that all entities that are involved in the execution must be locked. Equation (6.3) augments the definition of the action semantics $\psi_{a,x}$ (Equation (5.15)) in this sense.

$$\psi_{a,x} \mathrel{\widehat{=}} enabled_{a,x} \wedge [\![\boxed{\overline{ready} := \overline{false}}]\!]_V \; ; stat_{a,x} \tag{6.3}$$

where $\overline{ready}$ denotes the list of all $ready_j$ for $j \in E_{a,x}$ and $\overline{false}$ the list of *false* literals of the same length. The set V is the union of all state variables of the involved entities: $V = \bigcup_{j \in E_{a,x}} V_j$. The major difference between local and remote actions is the semantics of the statement that is executed.

Statement Semantics

The semantics of *local* actions remains as defined in Section 5.3.5, i.e. Equation (5.16). However, the semantics of the remote action is defined by the statement in the invoked interface. The semantics of the statement associated with an action is extended to include the semantics of remote actions:

$$stat_{a,x} \mathrel{\widehat{=}} (x \in X_a^{loc} \wedge stat_{a,x}^{loc}) \vee (x \in X_a^{rmt} \wedge stat_{a,x}^{rmt}) \tag{6.4}$$

The model does take into account the possibility of failure. Given that two entities are involved in the execution, the failure model of the remote action is more sophisticated than for the local action case. We differentiate two points of view. The viewpoint of the invoking agent and the viewpoint of the invoked object. Figure 6.1 depicts the different phases and the potential points of failure in the execution of a remote action. We use

i here to refer to the interface, viz. the viewpoint of the object, and x for the external action, viz. the viewpoint of the agent. a denotes the agent and o the object, $S_{o,i}$ the statement defined in the interface definition.

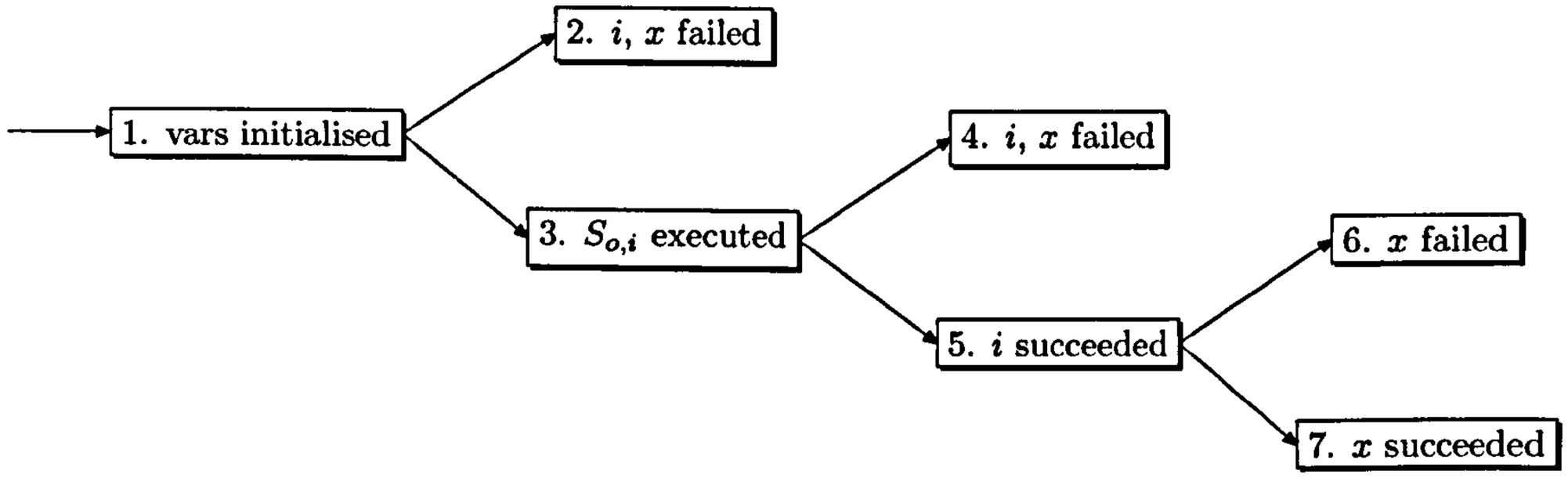


Figure 6.1: Schematic of remote action statement

1. Local variables are created and initialised. They represent local copies of the object variables that are modified by the execution of the statement, as well as the input parameters and output parameters of the invoked interface. The enforcement mechanisms then perform an authorisation check.

2. The execution failed due to the authorisation check. i, x indicates that the execution fails from both points of view, i.e. the failure is visible to the object *and* the agent[1].

3. The statement is executed. The execution of the statement requires time. If the specified timeout for the execution is exceeded, or an integrity check for the object fails, then the action will fail (4.). Otherwise the execution succeeds from the objects point of view (5.)

4. The execution failed due to a timeout or an integrity check made by the objects enforcement mechanisms. This failure is again indicated to both the agent and the object.

5. The statement execution succeeded and the object's state is updated. At this point the involvement of the object finishes and the object can be accessed again by other agents. Subsequently the agent will perform an integrity check on the output values.

[1]There are different opinions within the security community whether the access control failures should be indicated to the requesting party. Arguments for this approach are that the requesting side can take another course of action to cope with the failure. Some authors even provide a framework that gives more detailed information on *why* the access failed to allow the requesting party to obtain the required credentials for the execution. Arguments against providing information on failure is that the system becomes prone to attacks that use black-box analysis techniques that to develop a model of the system's security mechanisms in order to exploit covered channels.

6. The agent's integrity check failed. This means that the computation did not meet the agent's expectations. The failure is indicated *only* to the agent. Changes made to the object state are *not* rolled back. The integrity check is a foundation on which *trust* models can be built.

7. The agent's integrity check succeeded and the values of the output parameters are copied to the receiving agent variables. The execution is successful.

The semantics of the remote action statement is formalised in Equation (6.5).

$$\begin{aligned} stat^{rmt}_{a,x} \;\widehat{=}\; & \exists v'_0 \ldots v'_j, p_0 \ldots p_k, p'_0 \ldots p'_l, T \cdot \\ & (T = 1 \wedge T \text{ gets } T+1) \wedge \\ & (init_{i,x} \,;\, (([\![S'_{o,i}]\!]_{V^+} \,;\, (((succeed_i \wedge T < t_{max}) \,;\, (succeed_x \oplus fail_x) \\ & \qquad) \oplus fail_{i,x}) \\ & \quad) \oplus fail_{i,x})) \end{aligned} \tag{6.5}$$

The first line of the equation introduces local variables that store the temporary results of the computation (variables $v'_0 \ldots v'_j$) and the passed parameter values (input parameter: $In_i = \{p_0 \ldots p_k\}$, output parameter: $Out_i = \{p'_0 \ldots p'_l\}$). The state variable T models a local clock that is started with the statement execution. The behaviour of T is defined in the second line of Equation (6.5). It follows the formalisation of the different phases that have been depicted in Figure 6.1. The constant t_{max} denotes the time-out specified in the remote action definition. The set

$$V^+ \widehat{=} V \cup \{v'_0 \ldots v'_j\} \cup \{p_0 \ldots p_k\} \cup \{p'_0 \ldots p'_l\} \tag{6.6}$$

denotes the set of all state variables of the entities $e \in E_{a,x}$ that are involved in the execution of the remote action together with the local variables.

The different phases, that are depicted in Figure 6.1 are formalised in the following. Table 6.1 provides a short index to the definitions.

Local Variable Initialisation Local variables, representing a temporary copy of the object state, as well as the input variables of the interface are initialised.

$$init_{i,x} \widehat{=} [\![v'_0, \ldots, v'_k, p_0, \ldots, p_l \leftarrow v_0, \ldots, v_k, e_0, \ldots, e_l]\!]_{V^+ \setminus \{p'_0, \ldots, p'_l\}} \tag{6.7}$$

We denote the object variables that are modified in the statement by v_j, where $0 \leq j \leq k$ and $v_j \in V_{o,decl}$. Their local copies are denoted by v'_j. The set of input variables In_i of the interface as p_j, where $0 \leq j \leq l$ and $p_j \in In_i$. The corresponding values that are passed

Phase	Formalisation	defined in
1. vars initialised	$init_{i,x}$	(6.7)
2. i,x failed	$fail_{i,x}$	(6.8)
3. $S_{o,i}$ executed	$[\![S'_{o,i}]\!]^{+}_{V}$	(6.9)
4. i,x failed	$fail_{i,x}$	(6.8)
5. i succeeded	$succeed_i$	(6.10)
6. x failed	$fail_x$	(6.12)
7. x succeeded	$succeed_x$	(6.11)

Table 6.1: Phases in Remote Action Execution

by the agent during the invocation are denoted by e_j. During the initialisation the state of all participating entities is maintained. This is expressed by defining the semantics of the statement with respect to the variables in V^+ (defined in Equation (6.6)). The values of the output parameters $p' \in Out_i$ are not maintained by the initialisation.

Following the initialisation, the action either *fails* or continues with the execution of the statement S_i defined in the interface.

Remote Action and Interface Failure A failure at this point means that neither the object variables are modified (the viewpoint of the object), nor are the output parameters committed to the agent variables (the viewpoint of the agent). The failure is defined as:

$$fail_{i,x} \mathrel{\widehat{=}} [\![failed_{o,a,i}, failed_{a,x}, \overline{failed_{o,a,i}}, \overline{failed_{a,x}} \leftarrow true, true, \overline{false}, \overline{false}\ ; ready_o := true]\!]_V \tag{6.8}$$

The failure of an agent a's external action x, that invokes an interface i, is indicated in the agent by the control variables $failed_{a,x}$ and in the object by $failed_{o,a,i}$. Similar to the case of local actions in the previous chapter the complementary control variables $\overline{failed_{o,a,i}}$ and $\overline{failed_{a,x}}$ are set to *false*.

The local variables that store the temporary results of the computation are not maintained during the failure. This is required to be able to delay the failure due to a time-out. This is discussed in detail later in this section.

The agent and object variables remain unchanged. The lock on the object will be released in the final state. The lock on the agent must not be removed, as the agent immediately enters its deliberation phase.

Interface Statement Execution In case the access check was successful, the statement defined in the interface is executed on the local copies of the object variables.

$$S'_{o,i} \mathrel{\widehat{=}} S_{o,i}[v'/v] \tag{6.9}$$

$S'_{o,i}$ denotes the statement $S_{o,i}[v'/v]$ where every variable v is substituted by the corresponding local variable v'.

Interface Execution Succeeds (Object) After the execution of the statement, the action can again *fail* or *succeed* dependent on the outcome of an *integrity* check. This check is made from the object's point of view and represents assertions on the executed statement. If it fails, the action is considered to be failed from the object's *and* from the agent's viewpoint, viz. when the object discovers a fault, it will indicate this to the agent. This case is already formalised in Equation (6.8). The successful execution from the object's point of view is formalised as:

$$succeed_i \mathrel{\widehat{=}} [\![done_{o,a,i}, \overline{done_{o,a,i}}, v_0, \ldots, v_j \leftarrow true, \overline{false}, v'_0, \ldots, v'_j \; ; \; ready_o := true]\!]_{V+} \tag{6.10}$$

The successful execution of the interface is indicated by the control variable $done_{o,a,i}$, viz. agent a successfully executed object o's interface i. Similarly to the control variables for the agent only one of the variables $done_{o,a,i}$ or $failed_{o,a,i}$ can be *true*. This means that all other control variables are reset to *false*. The complementary set of control variables is denoted by $\overline{done_{o,a,i}}$, where $\overline{false}$ is the list of *false* literals of matching length.

The result of the computation (reflected in the local variables $v'_0, \ldots, v'_j$) is committed to the object's state. In the last state of the interval defined by $succeed_i$, the lock on the object is released by setting the control variable $ready_o$ to *true*. The specification $succeed_i$ represents only a success from the object's point of view, viz. the integrity constraints on the object are met. The action may only succeed if the time-out that has been specified in the invocation has not elapsed (conjunct $T < t_{max}$ in Equation (6.5), where t_{max} is the specified time limit).

Time-out The specification of the time-out in Equation (6.5) is not strict. It only states that no action may succeed if the time-out has elapsed, viz. every successful execution is performed within the time limit t_{max}. However, the maximal execution time is *not* bounded by the time limit. A concrete implementation can strengthen this requirement to ensure that t_{max} is a strict upper bound of the execution time. The following shows how the decision to fail can actually be delayed until the timeout does occur. The discussion uses the notion of refinement that is introduced in Chapter 9. However the example is simple enough to understand without referring to the concrete definitions.

The specification of failure in Equation (6.8) makes use of the temporal assignment to indicate failure, viz. the concrete time needed for the failure is not specified. This is used to delay the decision.

The statement S_i' is executed on the local variables and does not modify any of the variables that are controlled by $fail_{i,x}$. By rewriting the statement S_i to S_i' it is guaranteed that only the local variables v' are modified and all state variables in the set V remain stable. The specification of failure does *not* define the values for the local variables, and they can therefore assume any value over the interval. Consequently, by applying Theorem 9 (Chapter 9) we can introduce a finite prefix that maintains all agent variables:

$$fail_{i,x} \sqsubseteq (\mathsf{finite} \wedge ([\![\mathsf{idle}]\!]_V)^*)\ ;\ fail_{i,x}$$

a finite prefix can be introduced that maintains all variables in V. This prefix can be refined into any prefix interval of S_i'. The statement syntax guarantees that S_i' is finite.

$$\sqsubseteq f; fail_{i,x}$$

where f is a prefix of S_i' ($S_i' = f\ ;\ g$) and f does also refine the *idle* prefix (($\mathsf{finite} \wedge ([\![\mathsf{idle}]\!]_V)^*) \sqsubseteq f$). The latter is guaranteed by the substitution with local variables and the statement syntax.

This means that an implementation can delay the actual choice between the statement execution by executing f until the timeout does occur. If the time out is not reached, the chopping point is in the final state of the interval, viz. $g = \mathsf{empty}$. In this case the statement branch is chosen. Otherwise the chopping point is defined by the time out and instead of executing the suffix g, a failure is indicated by the execution of $fail_{i,x}$ choosing the fail branch of the first non-deterministic choice.

Remote Action Success (Agent) The successful execution from the viewpoint of the object does not necessarily mean that the action does succeed from the agent's point of view. Once the execution succeeded from the object's point of view, the agent verifies the results against its expectations that are expressed as integrity rules. If the results satisfy the constraints, the execution succeeds from the viewpoint of the agent. This is formalised as:

$$succeed_x \mathrel{\widehat{=}} [\![done_{a,x}, \overline{done_{a,x}}, e_0', \ldots, e_k' \leftarrow true, \overline{false}, p_0', \ldots, p_k']\!]_{V^+ \setminus V_o} \tag{6.11}$$

The successful execution is indicated to the agent by setting the control variable $done_{a,x}$ to *true*. $\overline{done_{a,x}}$ denotes again the complementary set of control variables. All output variables $e_0', \ldots, e_k'$ that have been specified in the external action are assigned to the results stored in the output parameters $p_0', \ldots, p_k'$. The set of variables that is controlled in this step excludes the object state variables, as the object can already be involved in another computation.

Remote Action Failure (Agent) If any of the integrity constraints are not met, then the execution fails. The failure affects only the agent and is formalised as follows:

$$fail_x \mathrel{\widehat{=}} [\![failed_{a,x}, \overline{failed_{a,x}} \leftarrow true, \overline{false}]\!]_{V^+ \setminus V_o} \tag{6.12}$$

The failure is indicated to the agent by setting the control variable $failed_{a,x}$; the list $\overline{failed_{a,x}}$ denotes again the complementary set of control variables.

Agent Semantics

The changes to the action semantics in the preceding Section 6.3.1 cover all modifications that are required to conservatively extend the semantics of an $SMAS_1$ agent to an $SMAS_2$. The extension does only affect the *execution phase* of an agent.

6.4 Single Object Semantics

Having defined the semantics of the execution of an external action by an agent, the semantics of a single object is straight forward. Let the semantics of a single object be defined by φ_o:

$$\varphi_o \mathrel{\widehat{=}} init_o \,;\, ((\bigoplus_{x \in X_o} \psi_{o,x}) \oplus [\![\mathsf{idle}]\!]_{V_o})^* \tag{6.13}$$

Here $init_o$ denotes the initialisation of the object's state and is subsequently defined in Equation (6.14). X_o is the set of all remote actions that invoke one of object's o's interfaces. The semantics of an interface invocation by the external action x is denoted by $\psi_{o,x}$ and subsequently defined in Equation (6.15). The exclusive-or guarantees that only one agent can access the object at a time. In the case that none of the agents in the system executes a remote action that invokes one of the interfaces of o, the object can stay *idle*. This behaviour is iterated (Chopstar).

6.4.1 Object Initialisation

The initialisation of objects is defined analogously to the initialisation of agents:

$$\begin{aligned} init_o \mathrel{\widehat{=}}\ & ready_o = false \wedge (\mathsf{keep}\ ready_o = false) \wedge \\ & \bigwedge_{a \in \mathcal{A}} \bigwedge_{x \in X_a} (done_{o,a,x} \leftarrow false \wedge failed_{o,a,x} \leftarrow false) \wedge \\ & \bigwedge_{v \in V_{o,decl}} (v \leftarrow I_v) \end{aligned} \tag{6.14}$$

Throughout the initialisation the control variable $ready_o$ remains *false*, preventing access to the object. All control variables $done_{o,a,x}$ and $failed_{o,a,x}$ are initialised to *false*. All object variables are initialised to their initial value I_v as provided in the object specification.

6.4.2 Interface Invocation

Every remote action invokes exactly one of the object's interfaces. The semantics of this invocation is denoted by $\psi_{o,x}$.

$$\psi_{o,x} \mathrel{\widehat{=}} enabled_{a,x} \wedge [\![\boxed{\overline{ready} := \overline{false}}]\!]_V \; ; stat_{o,x} \tag{6.15}$$

Here a denotes the agent executing the remote action x. Again $\overline{ready}{:=}\overline{false}$ sets the lock on all entities $E_{a,x}$ that are involved in the execution of x. Compare to Equation (6.3) on page (135). The semantics of the interface invocation differs only in the statement $stat_{o,x}$. The statement $stat_{o,x}$ is a prefix of $stat_{a,x}$ defined in Equation (6.5):

$$\begin{aligned} stat_{o,x} \mathrel{\widehat{=}} & \exists v'_0 \ldots v'_j, p_0 \ldots p_k, p'_0 \ldots p'_l, T \cdot \\ & (T = 1 \wedge T \text{ gets } T+1) \wedge \\ & (init_{i,x} \; ; (([\![S'_i]\!]_V \; ; (((succeed_i \wedge T < t_{max}) \\ & \qquad) \oplus fail_{i,x}) \\ &) \oplus fail_{i,x})) \end{aligned} \tag{6.16}$$

The semantics of the object interface execution does not contain the success or failure from the agent's perspective, enabling other agents to access the object, whilst the previously accessing agent is still performing the integrity check on the results of the computation.

The synchronisation between the object and the invoking agent is expressed by the fact that the semantics of the object's interface execution coincides with the semantics of the external action. An implementation, however, will not duplicate the behaviour, but instead keep one process idle whilst the other performs the computation. This may require the introduction of special marker variables to synchronise between both processes.

The following abstract example illustrates this. Let p_1 and p_2 be two processes executing in parallel. Let p_1 be the behaviour of an object and p_2 the behaviour of an agent. At some point in time, after a finite execution, the agent executes a remote action that invokes an interface of the object. f_2 denotes the behaviour of the agent executing the

remote action and f_1 the behaviour of the corresponding interface execution.

$$Sys = p_1 \wedge p_2$$
$$p_1 = \mathsf{finite} \,;\, f_1 \,;\, true$$
$$p_2 = \mathsf{finite} \,;\, f_2 \,;\, true$$

Assume that the finite prefix of p_1 and p_2 has the same length. In the SMAS semantics this is ensured by the use of $enabled_{a,x}$.

$$Sys = \mathsf{finite} \,;\, ((f_1 \,;\, true) \wedge (f_2 \,;\, true))$$

since f_1 is a prefix of f_2

$$f_2 = f_1 \,;\, g$$

we can substitute

$$Sys = \mathsf{finite} \,;\, ((f_1 \,;\, true) \wedge (f_1 \,;\, g \,;\, true))$$

Still each conjunct would behave like f_1 initially. To avoid the duplication the marker m is introduced:

$$Sys \sqsubseteq \mathsf{finite} \,;\, \exists m \cdot m = false \wedge (((f_1 \wedge (\mathsf{keep}\ m = false) \wedge \mathsf{fin}\,(m = true)) \,;\, true) \wedge ((f_1 \wedge (\mathsf{keep}\ m = false) \wedge \mathsf{fin}\,(m = true)) \,;\, g \,;\, true))$$

Whilst before both conjuncts could take a different amount of time to behave like f_1, the processes now synchronise on the marker variable m, viz. the execution of f_1 terminates in the same state for both processes. The behaviour f_1 of one of the processes is now redundant and can be eliminated.

$$= \mathsf{finite} \,;\, \exists m \cdot m = false \wedge (((f_1 \wedge (\mathsf{keep}\ m = false) \wedge \mathsf{fin}\,(m = true)) \,;\, true) \wedge ((true \wedge (\mathsf{keep}\ m = false) \wedge \mathsf{fin}\,(m = true)) \,;\, g \,;\, true))$$

Similarly to this abstract example, the synchronisation between a remote action as part of the behaviour of an agent and the interface semantics that is part of the object behaviour can be synchronised. The semantics of remote actions and their corresponding interfaces cannot be refined independently, as all design decisions made must preserve the

synchronisation between them.

6.5 Multi Agent Semantics

The semantics of the overall SMAS_2 system is a conservative extension of the SMAS_1 semantics. It combines the extended semantics of agents with the semantics of objects as follows:

$$[\![SMAS_2]\!] \mathrel{\widehat{=}} (\mathsf{halt}(\bigwedge_{a \in \mathcal{A}} term_a) \wedge \bigwedge_{a \in \mathcal{A}} (\varphi_a) \wedge \bigwedge_{o \in \mathcal{O}} \varphi_o) \tag{6.17}$$

Given the SMAS_2 specification the whole system could remain *idle* although the execution of an action is possible. It would be sensible to strengthen the specification to ensure that at least one action in the system is executed. In this case the additional conjunct *prog*, expressing a simple progression property, can be added to the SMAS_2 specification. The progression property is formalised as:

$$prog \mathrel{\widehat{=}} \Box \left(\left(\bigvee_{a \in A} \bigvee_{x \in X_a} enabled_{a,x} \right) \supset \left(\bigvee_{a \in A} \bigvee_{x \in X_a} \psi_{a,x} \right) \right) \tag{6.18}$$

In every suffix interval in which one of the actions in the system is enabled, one of the actions is executed. The semantics of the agents already ensures that any executed action is also enabled.

6.6 Example of SMAS_2

In this section the semantics of the motivating example provided in Section 6.2.4 is detailed and the behaviour of the agent and object explained on a state by state basis. The following listing is identical with Listing 6.3 on page 133 and is included here for the readers convenience.

Listing 6.4: Motivating Example for $SMAS_2$

```
1  /* Agent Definition */
2  agent ag :
3      /* Agent Variables */
4      var x = 0
5      var y = 0
6
7      /* Agent Actions */
8      when x < 2^32 - 1 do inc : ob.set(x+1, x)
9      when x < 2^31 do dbl : ob.set(2*x, x)
10
11     /* Deliberation */
12     deliberation : {
13     if x % 2 = 0 then inc,dbl← 1,0 else inc,dbl← 0,1 }
14 end
15
16 /* Object Definition */
17 object ob :
18     /* Object Variables */
19     var x = 0
20
21     /* Interfaces */
22     set (in a, out b) : {  x,b← a,a  }
23 end
```

The example introduces the object `ob` that encapsulates the variable `x` and provides the interface **set(in a, out b)** that will update the value of the object's variable `x` to the input parameter **a** and returns this value also as the output parameter `b`. The actions `inc` and `dbl` are now external actions that invoke the interface **set(in a, out b)** of `ob` with the appropriate new value. The key difference is that the variable `y` that is stored in the object can be accessed by other agents in the system via the object's interface.

The example is again kept simple and includes only a minimum of variables. The number of agents, objects, actions and interfaces, as well as the amount of information, that is transmitted in an interface invocation and the complexity of the computation that is performed are not providing any additional insights in the behaviour of the system and would only hinder the understanding of the mechanism. This example is only discussed at the specification level. The abstract behaviour of the system is depicted in Figure 6.2.

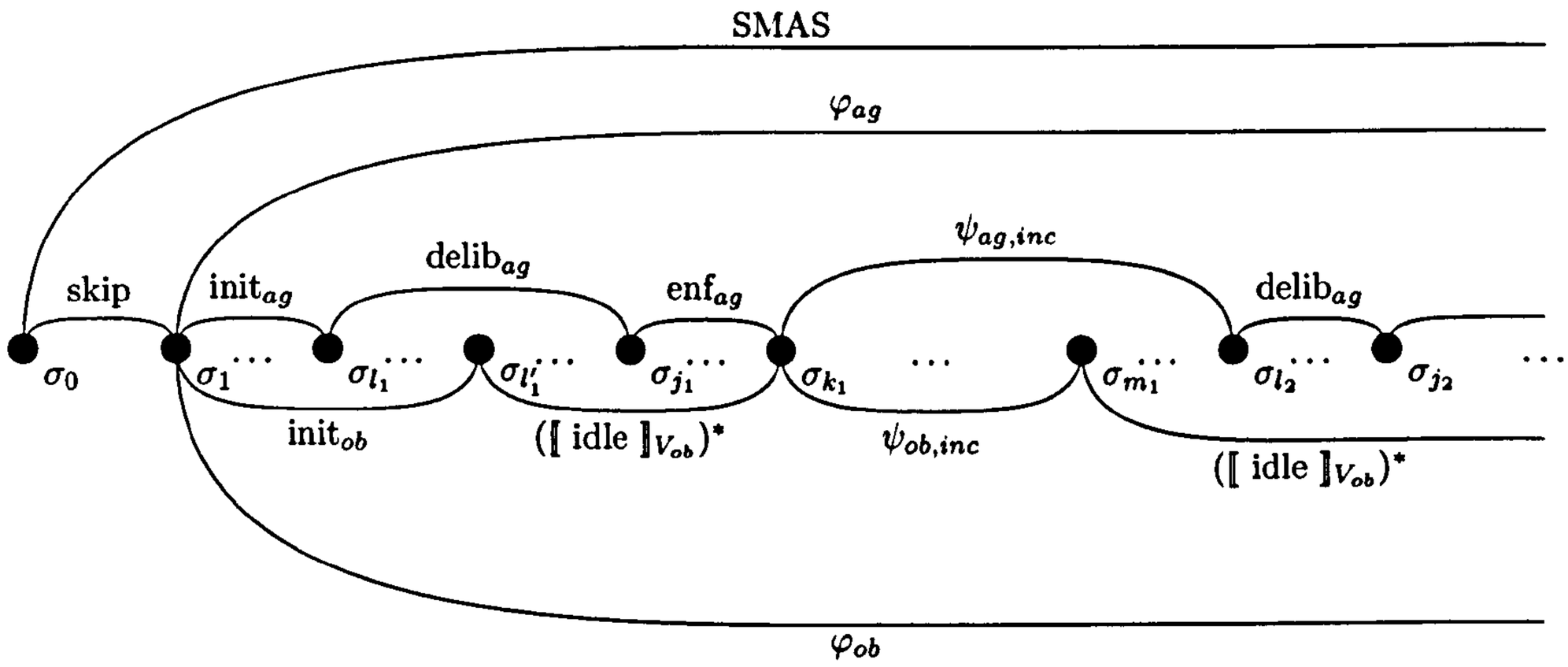


Figure 6.2: Abstract behaviour of the SMAS_2 example

Figure 6.2 depicts the agent's behaviour on the top of the state sequence and the object's behaviour below the state sequence. The behaviour of the agent is similar to the one shown in the example for SMAS_1. The object starts with its initialisation phase. The object initialisation completes in state $\sigma_{l'_1}$. The state $\sigma_{l'_1}$ is depicted *after* state σ_{l_1}. This is an arbitrary choice; it may as well be before or coinciding with the state σ_{l_1} dependent on the actual implementation of both initialisation phases. The object then remains *idle* as there is no enabled remote action for its interface `set`. This is because the agent is not yet *ready*.

After the agent finished the execution of its deliberation and enforcement phase in state σ_{k_1} the remote action `inc` is enabled, as the agent and the object are *ready* and the action's guard evaluates to *true* (because x=0 is an even number). The agent now behaves as defined by $\psi_{ag,inc}$ whilst the object will behave as $\psi_{ob,inc}$. Both behaviours are identical representing the synchronisation between the agent and the object until the state σ_{w_1} where the object finishes with the execution of $\psi_{ob,inc}$. It is ready again and remains *idle* until the next interface invocation. The agent continues with the behaviour $\psi_{ag,inc}$ and updates the agent's state, terminating in state σ_{l_2}. The agent then reenters its deliberation phase.

Table 6.2 shows the variables and their values in the depicted states of the execution. The "?" denotes that the value of the variable is not defined.

Agent Variables		σ_0	σ_1	σ_{l_1}	$\sigma_{l'_1}$	σ_{j_1}	σ_{k_1}	σ_{m_1}	σ_{l_2}	σ_{j_2}
x	Variable x	?	?	0	0	0	0	0	1	1
Object Variables										
y	Variable y	?	?	?	0	0	0	1	1	1
Priority Variables										
$\Pi_{ag,inc}$	Priority of `inc`	?	?	1	?	1	1	1	1	1
$\Pi_{ag,dbl}$	Priority of `dbl`	?	?	1	?	0	0	0	0	0
Control Variables										
$ready_{\mathtt{ag}}$	`ag` is not *ready*	?	?	F	F	F	T	F	F	F
$done_{\mathtt{ag,inc}}$	`ag` successfully executed `inc`	?	?	F	F	F	F	F	T	T
$failed_{\mathtt{ag,inc}}$	`ag` failed executing `inc`	?	?	F	F	F	F	F	F	F
$done_{\mathtt{ag,dbl}}$	`ag` successfully executed `dbl`	?	?	F	F	F	F	F	F	F
$failed_{\mathtt{ag,dbl}}$	`ag` failed executing `dbl`	?	?	F	F	F	F	F	F	F
$ready_{\mathtt{ob}}$	`ob` is *ready*	?	?	T	T	T	T	T	T	T
$done_{\mathtt{ob,ag,set}}$	`ag` successfully executed `set`	?	?	?	F	F	F	T	T	T
$failed_{\mathtt{ob,ag,set}}$	`ag` failed executing `set`	?	?	?	F	F	F	F	F	F

Table 6.2: Execution of the $SMAS_2$ Example.

The reason why the values of the priority variables are not defined in the state $\sigma_{l'_1}$ is that it lies within the agent's deliberation phase. The specification of this phase, however, does only guarantee that the priority values are defined in the final state of the phase.

The synchronisation between the agent and the object is given by the fact that both $\psi_{\mathtt{ag,inc}}$ and $\psi_{\mathtt{ob,inc}}$ start in the same state and that $\psi_{\mathtt{ob,inc}}$ is a prefix of $\psi_{\mathtt{ag,inc}}$.

6.7 Summary

The syntactic constructs for the specification of objects that represent the shared environment of agents have been introduced. The state of the environment is represented by the states of all objects in the system. Access to objects is controlled by interfaces that define computation on the state of the encapsulating object. These interfaces are invoked by agents in remote actions. The notion of local and remote actions provides a clear distinction between local computation and communication. The passing of parameters in remote actions enables agents to modify and sense their environment.

The execution of remote actions is more sophisticated than local actions as at least two system entities must synchronise for the communication. More entities also increase the number of points in which the execution can fail. If the remote action is authorised by the policy enforced by the invoking agent the execution can commence. Parameters are evaluated and passed to the invoked object. Based on the input parameters an authorisation policy, enforced for the object, can cause the remote action to fail. Similarly a time

out for the remote action may cause a failure. At the end of the execution the action may fail due to integrity constraints enforced for the object, or the result may violate integrity constraints enforced by the agent. Failure generally means that the agent and object variables are not modified and the failure is indicated in the control variables. However, the exception is that the failure due to an agent's integrity check does not affect the object. This means that the local decision of the agent to reject the results of the computation *does not* affect the environment and therefore can not be observed by other agents.

The behaviour of the $SMAS_2$ has been defined as a conservative extension to the semantics that was presented in the previous chapter. This means that the semantics of an $SMAS_2$ that does not contain objects and therefore does not contain remote actions is identical with the semantics of the $SMAS_1$ system. The semantics of the additional constructs has been described informally in Section 6.2 and was formalised in Section 6.3, 6.4 and 6.5. The example provided in the previous chapter has been modified and uses now remote actions that increment or double the value of a variable that part of the agent's environment. The example has been detailed and related to the formal definitions of the $SMAS_2$.

Chapter 7

Policies

Policies express access control, obligation and integrity requirements on agents and objects. This chapter introduces the syntax and semantics of SANTA policies. It shows how complex protection requirements can be expressed with ease using the expressiveness of policy rules and the composition of policies.

7.1 Introduction

A security policy expresses protection requirements on the system in a precise and unambiguous form. Policies in SANTA are concerned with access control, obligations and integrity. They relate to the entities in the system, and define constraints on their interactions. Access control requirements in this model are authorisation requirements, viz. constraints on the actions that a subject can perform on objects, or delegation requirements, viz. which subject can delegate which right to another subject. Obligation requirements express that subjects *must* perform specific actions. Integrity requirements define constraints on the effect that the execution of an action has on subjects and objects.

The aim of policies is to express these requirements at a high level of abstraction, hiding the details of the implementation that is necessary for their enforcement. In SANTA, policy rules are used as the basis for policy specifications. Rule-based languages are well established and well suited because most of these requirements are already informally expressed in the form of conditions and consequences. Each rule is expressed in terms of subjects, objects and actions. Subjects are the actors in the system. They can request access to objects, that represent the available resources. The term *action* is used to denote the mode of access. A typical example is the `-rwx` (read, write, execute) mode commonly used in UNIX systems to distinguish the mode of access to a file.

In SANTA two different types of policies are distinguished: *environmental* and *behavioural* policies. The former represents the more traditional view that the access to shared resources in the environment is protected. The latter addresses constraints on the behaviour of agents in the system.

Environmental Policies Environmental Policies protect the access to and the integrity of objects. For environmental policies *subjects* are SANTA agents. We assume that any user is represented by one or more agents in the SMAS. SANTA objects are resources in the shared environment of the agents — they are also the *objects* in the environmental policy. The actions that can be performed on an object are represented by the object's interfaces. Using the example from the previous chapter a policy could for example describe under what conditions the agent `ag` can invoke the interface `set(in a, out b)` on the object `ob`, or the constraints that must be met for the execution to be successful. Environmental policies can be enforced by *vigilant objects* and *security enforcer* mechanisms. They represent the traditional view of policies controlling the access to shared system resources. However, for agent systems another view is also of interest.

Behavioural Policies Behavioural Policies restrict the behaviour of subjects. *Subjects* are represented by agents. The behaviour of an agent is defined in terms of its actions by the deliberation phase, viz. the deliberation phase decides on the sequence of actions the agent executes. Agents can be complex software programs that could for example employ AI learning techniques to adapt to changes in the system or to the users preferences. These mechanisms are abstracted in SANTA in the deliberation phase of an agent. The functioning of the agent's decision making is not always transparent to the user. This is either due to the complexity involved or because modules that are provided by third parties are used. In this case it may be difficult for the user to *trust* the agent's reasoning. Whilst a user will undoubtedly appreciate *smarter* behaviour of the software agent most users will also be wary of the risks and the unpredictability that is involved. Behavioural policies define constraints that the user wishes to be adhered to. They are enforced by *vigilant agents* and can then provide some guarantees on the agent's behaviour. Behavioural policies do not protect the resources within the environment, but are restrictions on the behaviour of the agent itself.

For example: *My software agent has access to my bank account, which is represented by an object. The policy controlling access to the resources ensures that only my bank and the agent acting on my behalf have access to the account. However, anticipating that the reasoning engine of my software agent may be exploited by other malicious agents I want to constrain my agent to allow only 10 withdrawals of a maximum of 100 GBP a month.* This is a requirement that does not represent a constraint on the bank account, but a constraint on the behaviour of the agent.

A behavioural policy defines the conditions under which the agent can or must *choose* to perform a specific action. It also can place constraints that must be met before the agent accepts the result of an execution. For this type of policies the subject and the target of the execution is the agent. The actions that can be controlled are the agent's local and remote actions.

The distinction between environmental and behavioural policies is at the conceptual level. At the language level, the only difference is the subject, object and action pairing of a rule. This means with respect to the policy language the view of both policy types is uniform.

Policy Specification is the process of expressing informal protection requirements within the policy language. Policy rules form the basis of SANTA policy specifications and one of the major tasks during the specification process is the development of rules that adequately capture the informal requirements. The accurate specification of rules can be difficult when complex requirements need to be expressed. These are for example dependencies on the state of the system or dependencies on the history of the execution. The SANTA policy language provides support for both state and history dependencies.

Not all requirements can be easily expressed in form of rules. One example would be an electronic paper submission system that is staged in different phases, e.g. registration, submission, review, etc. During each phase a different policy applies, controlling the ability of users to submit, review and comment on a paper. This would require the policy to change dynamically at the transition from one phase to the next. This form of requirements is difficult and cumbersome to encode in rules, because the different phases have to be considered in each of the rules. The approach in SANTA is different in that these types of requirements are captured through policy composition.

Policy Composition The advantage of SANTA policies over the majority of other policy languages is that policies can be specified in small units, which are composed using a rich set of operators. The provided operators allow for policies to be composed along a temporal and structural axis.

Temporal composition leads to policies that change dynamically over time or on the occurrence of events. This can be used to specify the transition from one policy to another, for example to capture protection requirements for different phases.

Structural composition is concerned with the separation of subjects, objects, actions and policies, that apply only to a certain subset of these entities. A typical example is a hierarchical organisational structure, where each department defines its own policies. The combination of all these policies, together with some general protection requirements, yield the overall policy that applies to the organisation as a whole.

The concept of Dynamically Changing Access Control Policies has been recently investigated by Siewe [124]. Indeed, his work on temporal composition of authorisation policies forms the basis of the SANTA policy model. It has been significantly extended to capture other types of requirements, such as delegation, obligation and integrity. The structural composition and the unique problems that arise for the conflict resolution between two dynamically changing policies are addressed.

Using these new concepts, policies can be specified as compositions of smaller, simpler policies along both the temporal *and* structural axis. This compositionality is also of great advantage for the enforcement of policies. Whilst the overall composition allows for the

analysis of the system wide effects that the enforcement of the policy has, it is also a great aid to decompose policies into units that are enforceable by the mechanisms present in the system. This makes it possible to drop the assumption of a centralised enforcement mechanism and replace it by the notion of *vigilant agent*, *vigilant object* and *security enforcer*, that describe (de-)centralised enforcement.

The SANTA language constructs that can be used to specify policies rules and their compositions are described in Section 7.2. This section provides examples of how policy rules for authorisation, delegation, obligation and integrity can reference the current system state or the history of the execution. Section 7.4 describes the operators that can be used to compose policies along the temporal and structural axis. The specification oriented semantics of the policy language is given in Sections 7.3 and 7.5. Section 7.6 summarises the policy model presented in this chapter and highlights its advantages over other models.

7.2 Policies in SANTA

Policies in SANTA are an integral part of the system specification. They capture protection requirements for the SMAS. The smallest unit of a policy specification is a policy rule. Rules capture individual requirements, such as: "*allow administrators to change user passwords.*" The rule syntax and the informal semantics is detailed in Section 7.2.1. Sections 7.2.2 to 7.2.5 provide examples and the informal meaning of authorisation, delegation, obligation and integrity rules.

Rules are combined into larger units, named *simple policies*. Simple policies are a set of rules that are all enforced simultaneously. They define for example the protection requirements that apply in a specific situation or phase of the system execution. Simple policies are detailed in Section 7.2.6. This is followed by the informal description of the operators that can be used for the composition of policies along the temporal axis and the structural axis in Section 7.4.2.

Listing 7.1 shows the EBNF of the policy syntax. The individual constructs and their use are detailed in the subsequent sections.

Listing 7.1: EBNF for Policy Syntax

```
poldef = 'policy', id, ':' { funcdef }, policy.
policy = policy, ';', policy |
         'unless', expr, ':', policy |
         'aslongas', expr, ':', policy |
         intlit, ':', policy |
         'if', expr, 'then', policy, 'else' policy |
         'repeat', policy |
         policy, 'and', policy, ['deconflict', (spolicy | id)] |
         scope, policy |
         '(', policy, ')' |
         spolicy |
         id .
spolicy= '(', { rule }  ')' .
rule   = conseq, '(',id,',',id,',',id,')', 'when', f.
conseq = 'allow' | 'deny ' | 'decide' | 'oblige' | 'integrity'.
f      = f, ';', f | f, 'or', f | f, 'and', f |
         'sometimes', f | 'always', f | 'keep' f |
         'if', expr, 'then', f, ['else', f ] |
         ( intlit | listlit ), ':', f |
         ('exists'|'forall'), id, (('<',expr)|('in',expr)), ':' f |
         '(', f, ')' | expr .
scope  = 'scope','(',(list|'any'),',',(list|'any'),',',(list|'any'),')'.
funcdef= 'def' id, '(', [ id, {',', id} ], ')', ':', expr, 'end'.
id     = /* ... */ |
         'S' | 'O' | 'A' | 'T' |
         ['.'], 'subjects', ['.'] |
         ['.'], 'objects', ['.'] |
         ['.'], 'actions', ['.'] |
         ['.'], conseq, ['.'] '(',id,',',id,',',id,')' .
```

A policy definition is introduced by the keyword **policy**, followed by the system wide unique identifier of the policy. A policy definition can contain an optional set of function definitions. These can reference the system state and provide a way to abbreviate complex expressions for the use in the policy. The operators that can be used for composition are detailed in Section 7.4. The basic element of each composition is either a simple policy definition or the identifier of another policy. A simple policy is a set of policy rules enclosed in parenthesis. The set may be empty.

Each rule within the simple policy is introduced by a keyword, identifying its consequence. The consequences distinguish between the different types of rules that can be used in SANTA, e.g. authorisation rule, delegation rule, etc. Examples of these are given in Sections 7.2.2 to 7.2.5. The consequence is followed by a triple of identifiers that denote the subject, object and action to which the rule applies.

The premise of a policy rule is separated from the consequence by the keyword **when**. The premise can specify a past behaviour of the system that, when observed, leads to the consequence. This is detailed in Section 7.2.1. The keyword **T** is used to refer to the time that elapsed since the policy started to be enforced.

Each policy in SANTA is scoped, viz. the rules in the policy apply only to a specific set of subjects, objects and actions. These sets can be referenced in the policy specification using the keywords **subjects**, **objects** and **actions**. To avoid the overhead of having to define rules for each specific subject, object and action pairing, the keywords **S**, **O** and **A** can be used to reference free variables in the policy rule. They are bound by the scope of the policy, viz. **S** denotes any subject in the scope (**S** $\in$ **subjects**), **O** denotes any object in the scope (**O** $\in$ **objects**) and **A** denotes any actions in the scope (**A** $\in$ **actions**).

The keywords **subjects**, **objects**, **actions**, as well as the consequences can be prefixed or suffixed with a dot. The use of the dot notation is restricted to the parallel composition of policies and is detailed in Section 7.4.2.

7.2.1 Policy Rules

The rule-based approach to policy specification is advantageous because it provides a higher level of abstraction to the specification. An access control rule for example describes under what conditions a specific access control decision is taken — it does not define the actual mechanism that is used to *enforce* this decision. In this sense the policy is more abstract than a concrete check that is implemented directly in the accessing code. Also, the syntax of policy rules contains some high level constructs to reference the past behaviour of the system. Examples are the temporal modalities **always** and **sometime**.

Rule Structure Every rule consists of a premise and a consequence. The premise describes the condition that when observed by the enforcement mechanism leads to the specified consequence. The general form of a rule is:

> *consequence* **(x,y,z) when** *premise*

The *consequence* of a rule distinguishes the class of requirements that can be expressed in that rule. The triplet (x,y,z) references the subject, object and action to which the rule applies. Finally, the premise of the rule describes the condition under which the rule fires.

Policy Scope Every policy has a scope, that defines to which subjects, objects and actions the policy applies to. The scope of a policy is accessible in the syntax using the keywords **subjects**, **objects** and **actions**, that represent the set of subjects, the set of objects and the set of actions in the scope of the policy. The scope of a policy affects the rules contained in that policy. The subject, object and action in each rule definition must be within the scope of the policy (**x** $\in$ **subjects**, **y** $\in$ **objects** and **z** $\in$ **actions**).

Rules can also be defined in terms of *all* subjects, objects and actions in the scope using the keywords **S**, **O** and **A**. This provides a greater flexibility in the expression of

rules. By default the scope is the universal scope, viz. the sets of all subjects, objects and actions in the system.

Using the keywords S, O and A Often rules apply to more than one subject, object or action. To avoid the effort to duplicate the rules for the different subject, object and action pairings, the keywords **S**, **O** and **A** can be used to reference free variables in the rule definition. These free variables are bound by the scope of the policy in the semantics of each rule. For example, given that the scope of the policy is:

$$\textbf{subjects} = \{x_1, \ldots, x_n\} \quad \textbf{objects} = \{y_1\} \quad \textbf{actions} = \{\texttt{read}\}$$

The requirement to unconditionally grant all subjects in the scope of the policy the right to perform `read` on object o_1, would require the definition of n rules:

```
allow (x1,y1,read) when true
allow (x2,y1,read) when true
/* ... */
allow (xn,y1,read) when true
```

Using the keyword **S** this could be written more compact as:

```
allow (S,y1,read) when true
```

Referencing state and history in rules Rules in SANTA can express *state* and *history-based* dependencies. This is achieved by allowing the premise of a rule to reference the current state of the system or the behaviour of the system in the past. The referenced state and behaviour is restricted to the part of the system that is observable by the mechanism enforcing the policy. This means that policies and the mechanisms enforcing them cannot be seen in separation if the requirements are dependent on the system state or the history of the execution. The following is a brief overview of the constraints for the different enforcement mechanisms:

Vigilant Agent A vigilant agent can enforce behavioural policies. Rules that are enforced using this mechanism can reference the agent variables as well as the control variables **done(x)** and **failed(x)** that indicate the success or failure of the action **x**.

Vigilant Object A vigilant object can enforce environmental policies. Rules that are enforced using this mechanism can reference the object variables as well as the control variables **done(x,y,z)** and **failed(x,y,z)** that indicate the success or failure of object **y**'s interface **z** by subject **x**. Parameters of the interfaces can also be referenced.

Security Enforcer A security enforcer can enforce environmental policies. Rules that are enforced using this mechanism can reference variables defined in the security enforcer, as well as the control variables **done**(x,y,z) and **failed**(x,y,z) for the objects under its protection. Parameters of the interfaces can also be referenced.

These constraints are *not* a limitation of the policy language itself. Policies *can* violate these restrictions. However, it is beneficial to be aware of the limitations of the enforcement mechanisms when writing the policy to ensure that it is enforceable.

Premise of Policy Rules

The premise of a rule allows the expression of a set of behaviours, that when observed trigger (or fire) the rule. The syntax of the premise is restricted to ensure that the rules are implementable. Due to the expressiveness of the model care needs to be taken to ensure that the premise does capture the requirement correctly and does not define a too large (or too narrow) set of behaviours that trigger the rule. Figure 7.1 depicts informally the relation between the set of behaviours expressed in the premise and the consequence.

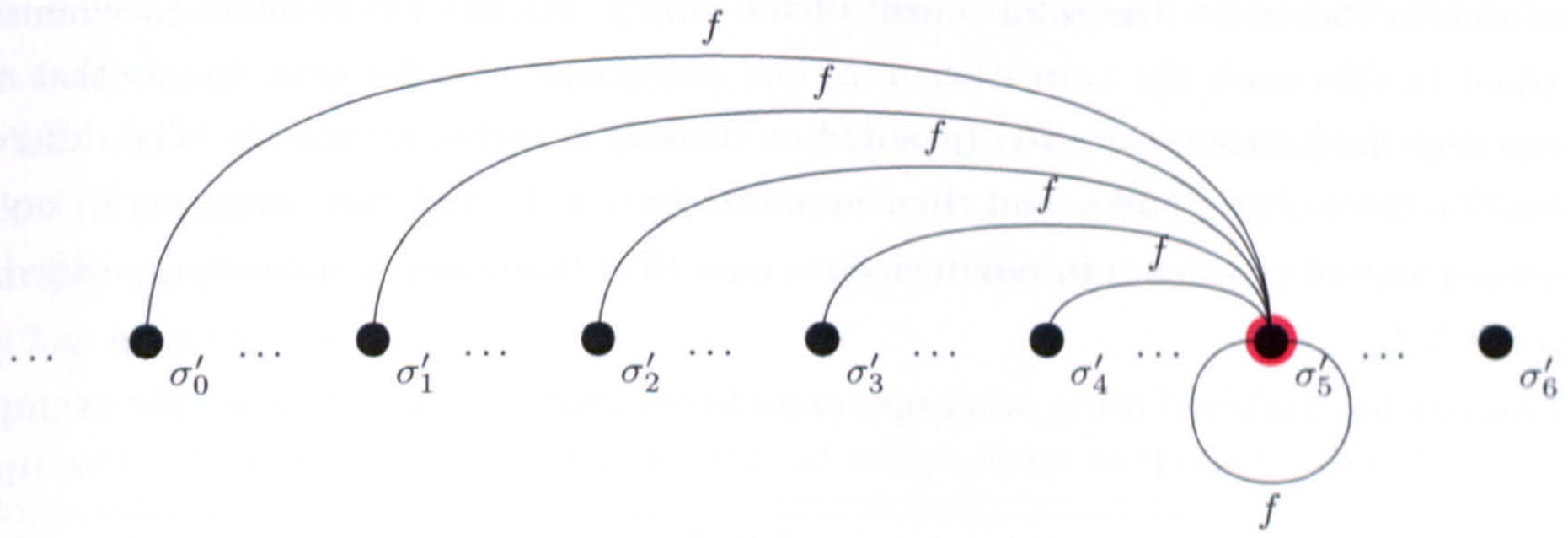


Figure 7.1: Informal Interpretation of a Policy Rule

Assume the policy containing the rule started to be enforced in state σ'_0. Given that the premise of a rule is represented by the formula f and the consequence of the rule is evaluated in the state σ'_5, then the consequence is *true* if the behaviour described by f is satisfied by *at least one* of the intervals depicted in Figure 7.1. For example the rule `consequence when x=0` would mean that if x has been zero in state σ'_0, or $\sigma'_1, \ldots,$ or σ'_5 then the consequence would be *true* in state σ'_5.

The expressiveness of the policy model is both a boon and a bane. The benefits are that many complex requirements can be captured in a concise and short form. The disadvantage is that some requirements, that are straightforward to express in other policy models, require more thought. This is especially the case when rules depend on the past

behaviour of the system. The following illustrates the syntax that can be used to specify premises of policy rules that are state or history-dependent.

Constraining the length of the behaviour Often rules are state-dependent, this means that f does not actually express a behaviour, but rather a predicate on the current state of the system. For example to express that the rule should only fire if x is *now* (in state σ'_5) equal to zero then the length of the formula f must be explicitly restricted by writing:

```
consequence when 0 : x=0
```

It is also possible to specify a list to restrict the length of the behaviour, for example the rule:

```
consequence when [0..2] : x=0
```

would fire if $x = 0$ holds in σ'_5, σ'_4 or σ'_3, viz. on an interval of length 0, 1 *or* 2 in the past. The concrete meaning of 2 states in the past depends on the concrete enforcement mechanism that is chosen for the enforcement of the policy. For the enforcement mechanisms described in this work the transition from one policy state to the next means that the enforcement mechanisms solicited (granted or denied) exactly one access. The different abstraction level of the policy and the computational model and their mapping is key to the refinement of policies into enforcement code. The mapping is formally specified in Section 8.5.2.

This can be combined using other operators in the syntax of the premise. For example:

```
consequence when (true ; ([0..2] : x=0)) and (3: x>0)
```

would express that in one of the last three states x was zero and in state σ'_2 the value of x was positive.

Given this form of constraining the time over which the premise of the rule should be evaluated, it is obvious that some requirements cannot be expressed. One example is the requirement that "the rule should fire if x was *true* when the policy started to be enforced." This type of requirement demands knowledge of the history with respect to to the enforcement. In SANTA the special keyword T has been introduced, that represents a local clock that is started with the enforcement of each policy rule. T references the enforcement time of the rule. This allows to express the above requirement as:

```
consequence when T : x=0
```

Another type of requirement where the enforcement time **T** is required is typically the informal meaning of "the rule should fire if **x** was zero 3 states in the past". Of course this can be readily expressed as:

```
consequence when 3 : x=0
```

This would restrict the interval to length 3. The problem with this solution manifests itself in the beginning of the enforcement. Assume that the rule is to be enforced in state σ'_1. In this case there are only two intervals that can be checked whether they satisfy the premise: the unit interval $\sigma'_0\sigma'_1$ and the empty interval σ'_1. None of them can satisfy the requirement that the length of the interval is 3. Often these types of requirements imply that if there is not enough history available then the requirement should hold for the available history. This lenient interpretation of the example can be expressed as:

```
consequence when [T..3] : x=0
consequence when 3 : x=0
```

This means that in the initial state the premise is evaluated over any interval spanning from length zero to length 3, in the second state of the enforcement over any of the intervals spanning from length 1 to length 3 until finally from the fourth state onwards only the past intervals of length 3 are considered[1].

The use of the construct **[0..T]** is discouraged, as is the omission of the length specification (it is easy to see that **x = 0** in the premise is the same as **[0..T] : x = 0**). Whilst in principle computable, the evaluation of the rule is increasingly costly the longer the policy rule has been enforced.

Sequence It is possible to use the sequential composition in the premise of rules:

```
consequence when 1:x=2 ; 1:x=1 ; 0:x=0
```

This rule fires if **x** is *now* zero, in the previous state it was one and in the state before **x** had the value two. This is similar to writing :

```
consequence when 2:x=2 and (true ; 1:x=1) and (true ; 0:x=0)
```

but provides a more convenient form to express the sequence, especially if time ranges are specified within a sequence.

Temporal Modalities Often the modalities **always** or **sometime** are used informally in requirements. These can be directly expressed in SANTA. Their scope is restricted by the length of the interval specification that contains the modality. For example writing:

[1]The list production **[i..j]** results in a list of integers k such that $j \leq k \leq j$. It produces the empty list, if $i > j$.

```
consequence when 2: sometime x=0
```

Denotes within the past interval length two there is some suffix interval that satisfies $x = 0$. In other words it means that:

```
consequence when 2: (x=0 or (1:true ; x=0) or (2:true ; x=0))
```

The global length specification is important, to limit the interpretation of the behaviour to the past interval of length 2. The case for **always** is similar, however all suffix intervals must satisfy $x = 0$, which is equivalent to replacing the **or** in the above rule with an **and**.

The informal meaning of the conditional choice is self explanatory. The existential and universal quantification can be used to quantify over lists, e.g. the list **subjects**, **objects** and **actions** that can be used to access the policy scope.

Consequence of Policy Rules

Authorisation, Delegation, Obligation and Integrity rules are distinguished syntactically by their consequence. Table 7.1 provides an overview of the available consequences:

Authorisation (see Section 7.2.2)	
allow (x,y,z)	positive authorisation
deny (x,y,z)	negative authorisation
decide (x,y,z)	decision rule
Delegation (see Section 7.2.3)	
allow (x,y,**deleg**(x1,x2,y1,z1))	positive delegation
deny (x,y,**deleg**(x1,x2,y1,z1))	negative delegation
Obligation (see Section 7.2.4)	
oblige (x,x,z)	obligation
Integrity (see Section 7.2.5)	
integrity (x,y,z)	integrity

Table 7.1: Consequences in Policy Rules

7.2.2 Authorisation Rules

Authorisation rules express access control requirements. Three different types of rules are concerned with authorisation: *positive* authorisation, *negative* authorisation and *decision* rules.

Positive Authorisation Positive authorisation rules are statements that indicate under which condition an access request should be granted. It is important to note that it is

only an *indication*, which is taken into account for the final access decision of the policy. Listing 7.2 provides some examples of positive authorisation rules.

Listing 7.2: Example Positive Authorisation Rules

```
1  allow (S, O, A) when true
2  allow (S, passwd, write) when 0: group(S, admin)
3  allow (S, O, A) when T: (
4        exists y in objects : exists z in actions :
5        ( (dataset(O) = dataset(y)) and
6          (sometime done(S,y,z)) ) )
```

The first rule states an unconditional positive authorisation for all subjects, objects and actions in the scope of the containing policy. This is an example of an activity based authorisation rule.

The second rule states the positive authorisation for any subject in the policy scope that is a member of the group `admin`. Here the predicate `group(subject, group)` denotes the group membership test. This relation is maintained by the system and must be accessible to the mechanisms that enforce the policy. The preceding `0:` denotes that this condition must be fulfilled at the time of the access control check. This is an example of a state-based authorisation rule.

The third rule states a positive authorisation from the Chinese Wall policy [35]:

> Once a subject [denoted by **S**] has accessed [denoted by **z**] an object [denoted by **y**], the only other objects [denoted by **O**] accessible by that subject are within the same company data-set [denoted by `dataset`] ...

The preceding **T:** denotes that the length of the subsequent behaviour is the time since the rule started being enforced. The rule is actually more precise, as it explicitly specifies that **O** and **y** must have been in the same data-set at the time the rule started being enforced. The **sometime done(S,y,z)** denotes that at some point in time since the enforcement of the rule the subject **S** successfully performed an action **z** on an object **y** in the same data-set. This rule is an example of a history-based authorisation rule.

Negative Authorisation Negative authorisation rules are statements that indicate under which condition an access request should be denied. Similarly to positive authorisations, they are only an *indication*, which is taken into account for the final access decision of the policy. Listing 7.3 shows examples of negative authorisation rules.

Listing 7.3: Example Negative Authorisation Rules

```
1  deny  (S, passwd, write) when 0: group(S, user)
2  deny  (S, O, read) when 0: (level(O) > clearance(S))
```

The first rule denies all subjects in the group `user` to `write` to the object `passwd`. The second rule is the *no read up* rule from the Bell-LaPadula policy [86]. It states that no subject with a clearance level (denoted by `clearance(S)`) that is lower than the security level of the object (denoted by `level(O)`) is allowed to `read` information.

Decision Rules and Conflict Resolution Decision rules specify the final access control decision of a policy. Any policy should contain at least one decision rule, as otherwise no access will be granted by the policy. The alternative term "conflict resolution rule" originates from the fact that this rule de-conflicts the policy if a positive and negative authorisation is derived for a specific access. The term *decision rule* describes more accurately the fact that *any* access control decision is defined by the policy is decided by one or more of these rules — not only decisions in the conflicting case. Listing 7.4 provides three widely used decision rules.

Listing 7.4: Example Decision Rules

```
1  decide (S,O,A) when 0: allow(S,O,A)
2  decide (S,O,A) when 0: not deny(S,O,A)
3  decide (S,O,A) when 0: allow(S,O,A) and not deny(S,O,A)
```

The first rule states that access is granted if a positive authorisation can be derived from the policy. This rule is used in *closed policies*, where any access is denied, unless it is explicitly allowed. The rule ignores all negative authorisation rules, viz. negative authorisation rules in a policy with this decision rule are not having any effect on the policy decision and should be omitted.

The second rule states that access is granted if no negative authorisation can be derived from the policy. This rule is used in *open policies*, where any access is allowed, unless it is explicitly denied. Blacklists are typical examples of open policies. The rule ignores all positive authorisations, viz. positive authorisation rules in a policy with this decision rule are not having any effect.

The last rule in Listing 7.4 states a rule that is used in *hybrid policies*. Hybrid policies are taking into account both, positive and negative authorisation. Hybrid policies are suitable to express more complex access control requirements. The difficulty with hybrid policies is that conflicts can occur, in the sense that a subject is at the same time allowed and denied to access a resource. In this cases the decision rule does also resolve the conflict. The rule in the example states that access is only granted if explicitly allowed and not explicitly denied in the policy. It therefore gives precedence to denials.

It is allowed to have more than one decision rule in the policy. However, it is important to note that it is sufficient to have *one* decision rule firing for the access to be granted. Like other rules, the decision rules can also contain state and history-dependent premises.

7.2.3 Delegation Rules

Delegation in this work means the delegation of rights, only. Delegation of duties is not considered. Delegation rules define which subject is allowed to delegate which rights to which other subjects. The delegating subject is referred to as the *delegator* (or also *grantor*) and the target subject of the delegation as the *delegatee*. The delegated right here denotes the right to execute an action on an object.

Delegation rules are modelled as specialised authorisation rules, where the controlled actions are `delegate` and `revoke`. The object of the delegation action is dependent on the concrete implementation of the delegation mechanism. In this work the target object is required to be the same system entity that implements the enforcement mechanism that controls the access to the object. This means either the object, in case of vigilant enforcement or the security enforcer that is controlling the access.

In general the only assumption is that all enforcement mechanisms that control the access to the object for which the right has been delegated can determine whether the right has been delegated or not. This can be implemented by simply requiring the object of the delegation to be the enforcement mechanism itself. Alternatively a trusted third party authority could be the object for the delegation action, if a certificate based implementation for delegation is used. This work constrains itself to the first, simpler version, although undoubtedly certificate-based models have the advantage of better scalability in open environments.

Listing 7.5: Delegation and Revocation

```
1  /* Check if currently delegated */
2  def deleg(g, d, y, z) : [g,d,y,z] in delegations end
3  /* Interfaces for delegation */
4  delegate(in g, in d, in y, in z) : {
5    var set← false : {
6      for i < |delegations| : {
7          if (delegations[i] = [] and not set) then {
8              delegations[i]← [g,d,y,z];
9              set← true
10         }
11     } ;
12     if (not set) then {
13       delegations← delegations + [g,d,y,z]
14     }
15   }
16 }
17 revoke(in g, in d, in y, in z) : {
18   for i < |delegations| : {
19       if (delegations[i] = [g,d,y,z]) then
20           delegations[i] := []
21   }
22 }
```

The effect of a successful delegation is denoted by the predicate `deleg(g,d,y,z)` with the meaning that the delegator `g` has delegated the right to perform action `z` on object `y` to the delegatee `d`, viz. `deleg(g,d,y,z)` is *true* if the right has been delegated. A successful revocation means that the predicate `deleg(g,d,y,z)` is *false* from thereon. This is formally described by the following two SANTA interfaces that every vigilant object and security enforcer implements. The enforcement mechanism must maintain local data structures to keep track of the current delegations. The listing above assumes that delegations are stored in a list `delegations` that contains the delegation quadruples. Listing 7.6 provides small examples of delegation rules.

Listing 7.6: Example Delegation Rules

```
1 allow (S,O,delegate(g,d,y,z)) when 0: S = g
2 allow (S,O,revoke(g,d,y,z)) when S = g and sometime done(d,y,z)
3 allow (S,O,revoke(g,d,y,z) when 0: group(S, admin)
```

The first rule states that any subject `S` is allowed to delegate any right from itself to any other subject `d`. The target `O` of the delegation is in this case not explicitly specified, but could for example be the object `y`. This would mean that the delegator notifies the object's vigilant enforcement mechanism directly of the delegation. The second rule states that the delegator of a right is only allowed to revoke the right it delegated if the delegatee has at some point in time successfully exercised this right. The third rule shows an example where a subject `S` can revoke rights that have been delegated to other subjects `d` if `S` is a member of the group `admin` at the time of the check.

Effect of delegations

Access rights are delegated with the intention to affect access control decisions. Delegation rules, however, do only define who can delegate which right to whom. The effect that delegation has on authorisation is modelled as authorisation rules, that depend on the delegation predicate `deleg(g,d,y,z)`. Listing 7.7 shows several examples of how a delegation can affect authorisation decisions.

Listing 7.7: Effects of delegations

```
1 allow  (S,O,A) when 0:
2        ( exists g in subjects : deleg(g,S,O,A) )
3 decide (S,O,A) when 0:
4        ( exists g in subjects : deleg(g,S,O,A) and
5          not deny(S,O,A) )
6 decide (S,O,A) when 0:
7        ( exists g in subjects : deleg(g,S,O,A) and
8          not deny(S,O,A) and not deny(g,O,A) )
```

The first rule states that a positive authorisation can be derived for any subject **S** from the fact that a subject **g** delegated the right to it. This does not necessarily ensure that **g** does hold the right itself. Additionally the authorisation decision is also dependent on the decision rule in the policy.

The second rule is a decision rule that explicitly grants the access in case the right has been delegated to subject **S** and **S** is not explicitly denied to exercise this right. Similarly to the first rule this rule does not take into account the authorisation of the delegator at the time of the access.

The third rule is very similar to the second,but includes am additional check whether the delegator is currently permitted to exercise the delegated right himself.

These examples show the diversity of different notions of what it actually means to delegate a right. The fact that the effect of a delegation on the access control decision is made explicit is highly beneficial, as there is no definite agreement in the community of what a delegation actually means and how the access control decisions of the system are affected by it.

7.2.4 Obligation Rules

Obligation rules state under which condition a subject *must* perform a specific action. Given the autonomy of agents in the model, obligations cannot be enforced externally, viz. an agent that is engaged in some computation cannot be interrupted to comply with its obligations. Consequently obligations can only be enforced if they are defined as behavioural policy rules. Using the bank example, that was used to motivate behavioural policies, the owner of an agent could define the obligation to notify the owner whenever it withdrew money from the bank account. This would be expressed by the rule in Listing 7.8:

Listing 7.8: Example Obligation Rule

```
1 oblige (S, S, notify) when 0: done(S, S, withdraw)
```

This assumes that the agent, does define the actions `withdraw` and `notify`. Obligations were the main motivation to introduce priorities in the action selection process of an agent. The intuition is that an obligation in a behavioural policy can overrule the decision made by the agent in its deliberation phase. The owner of the agent can therefore force the agent to the execution of an action using obligations. However, even if the agent is obliged to perform a specific action it is not necessary the case that it does execute the action. The following list describes the cases where the action is not executed:

1. synchronisation requirements with other system entities (for example in the case of *remote actions*) prohibit the execution.

2. the obligation does conflict with the authorisation, viz. the obliged action is not permitted.

3. the obligation conflicts with the action's precondition, viz. the obliged action cannot execute.

4. the obligation does conflict with another obligation. The agent has two different obligations simultaneously. In this case the choice between the obliged actions is non-deterministic[2].

7.2.5 Integrity Rules

Integrity rules in policies provide a far more flexible way to define assertions on the *correct* execution of the system. Traditionally assertions are checks whether the state of the system satisfies a specific constraint, typically expressed as a boolean expression. If the constraint is satisfied the execution commences, otherwise the system stops. Other forms of integrity checks are less rigid and indicate the failure through error messages and log entries.

Most integrity checks that can be expressed in other languages are not able to distinguish between the users that are actually executing the action. In SANTA integrity constraints can differentiate between users. For example: *In a system, where integrity constraints define the maximum allowed deviation of the results from experienced values, there may be different threshold for different users. Systems that use statistical methods typically need to be calibrated by a dedicated system maintainer. During the calibration some of the results deviate more than would be deemed acceptable during normal operation. In this case there should be different integrity constraints for a system maintainer than for a normal user.*

Another advantage of SANTA integrity constraints is that rules allow not only to access the current state of the system, but also the *history*. This means that an integrity rule could state that the results from the action execution are in a specific relation to results that have been previously computed. A good example is the computation of Fibonacci numbers. Listing 7.9 shows the definition of a SANTA object that computes Fibonacci numbers.

The variable `x` is initialised to a list containing the first two Fibonacci numbers. The interface `get(out n)` can be invoked by agents to obtain the next number in this sequence.

[2]It is in principle possible to deconflict obligations along the line of the conflict resolution between positive and negative authorisations. However, as the number of potential obligations is determined by the number of actions the agent does encapsulate, the conflict resolution rules will be more complex. It is advisable to check the policy for conflicting obligations by analysing whether the premises of two different obligations can be satisfied simultaneously. It is also of interest to show that the obligation is authorised, by checking against the premise of the decision rules.

The statement in the interface definition states that the return value is the first element of the list.

Listing 7.9: Example Integrity Rule

```
1  object fibo :
2    /* Fibonacci sequence */
3    var x = [0,1]
4
5    /* Returns the Fibonacci sequence */
6    get(out n) {
7      n,x[0],x[1] ← x[0],x[1],x[0]+x[1]
8    }
9  end
```

It also shifts the Fibonacci number stored in the second list element to the first position and computes the next number in the sequence. To use this example to check the integrity of the object `fibo` a random error is introduced in the statement, by rewriting line 7 in Listing 7.9 as:

```
7  n,x[0],x[1] ← x[0],x[1],if (random(1)=1) then x[0]+x[1]
8                                           else x[1]-x[0]
```

Of course we assume that in a real-world application, the faults are not introduced on purpose and are a result of mistakes that are made by the programmer. An integrity policy can be used to check whether the computation satisfies certain constraints. Typically the constraints that are expressed are simpler than the actual computation itself and represent only some of the necessary constraints. For the computation of Fibonacci numbers this could be for example that it is a monotonic increasing function. Listing 7.10 shows an integrity policy that captures just this:

Listing 7.10: Example Integrity Rule

```
1  policy vo_fibo :
2  (
3    integrity (S, fibo, get) when 0:( x'[0] >= x[0] and x'[1] >= x[1] )
4  )
5  end
```

The prime notation is used to reference the new value of the variable. The above states that the new value of each list element must be greater or equal to its previous value. This is a policy that references the internal state of the object, which limits the mechanisms that can be used for its enforcement to the *vigilant object* mechanism.

The effect of enforcing this policy would be that any computation that would try to assign a list element to a value that is less than its previous value would fail. As a result

of the failure, the list values would effectively remain unchanged. In the example above it would mean that every time the **else** branch of the conditional expression is chosen, the action would fail. The exception to this is if the **else** branch is chosen for the first execution of **get**. In this case the new value for **x**[1] would be assigned to 1 $(1 - 0)$, which is incidently the correct number in the sequence.

Of course the integrity rule could be specified more accurately to amount to the exact specification of the interface. More interesting is the case where the integrity policy is defined solely in terms of the object's interface, viz. not requiring any knowledge of the internal state of the object. This kind of policies can be enforced by security enforcer mechanisms, viz. externally. Listing 7.11 shows an example of an integrity policy that accurately defines the constraints for the Fibonacci sequence.

Listing 7.11: Example Integrity Rule

```
1  policy se_fibo (any, [fibo], [get]) :
2  (
3    integrity (S,fibo,get(n)) when T:
4      always (forall s in subjects : (not done(s,fibo,get)))
5      and n = 0
6
7    integrity (S,fibo, get(n)) when T:
8    ( (keep (forall s in subjects: (not done(s,fibo,get))));
9      (1:done(S,O,get)) ;
10     (keep (forall s in subjects: (not done(s,fibo,get))))
11   ) and n = 1
12
13   integrity (S,O, get(n)) when
14   ( 1:done(S,O,get(n0)) ;
15     (keep (forall s in subjects: (not done(s,fibo,get)))) ;
16     1:done(S,O,get(n1)) ;
17     (keep (forall s in subjects: (not done(s,fibo,get))))
18   ) and n = n0 + n1
19 )
20 end
```

The specification contains three different rules that apply to three special cases in the execution. The first rule states that if the interface **get** has never been invoked before, then the returned value must be equal to zero. More accurately the rule reads that if over the past **T** states (viz. since the policy is enforced) it has always been the case that no subject successfully executed the interface **get** then the return value must equal zero. This is depicted in Figure 7.2.

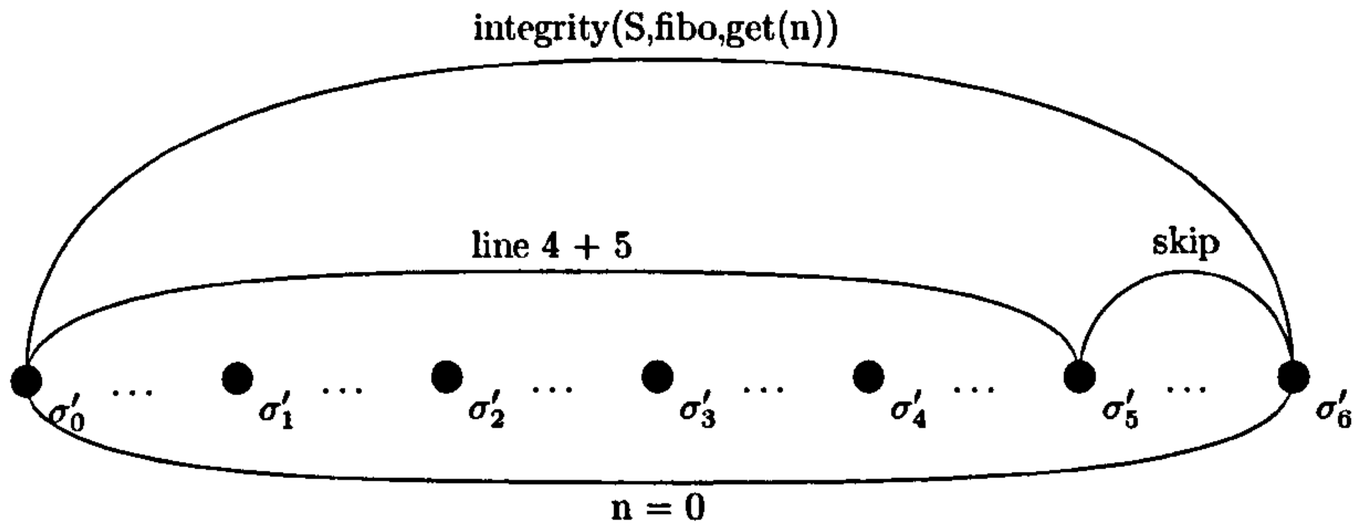


Figure 7.2: 1. Integrity Rule: No previous access to `get`

The second rule states that if the interface `get` has been invoked exactly once before, the returned value must be equal to one. More accurately it means that the interval describing the behaviour since the policy enforcement started, can be decomposed into two intervals in such a way that the interface `get` has only been successfully executed in the shared state. This is depicted in Figure 7.3.

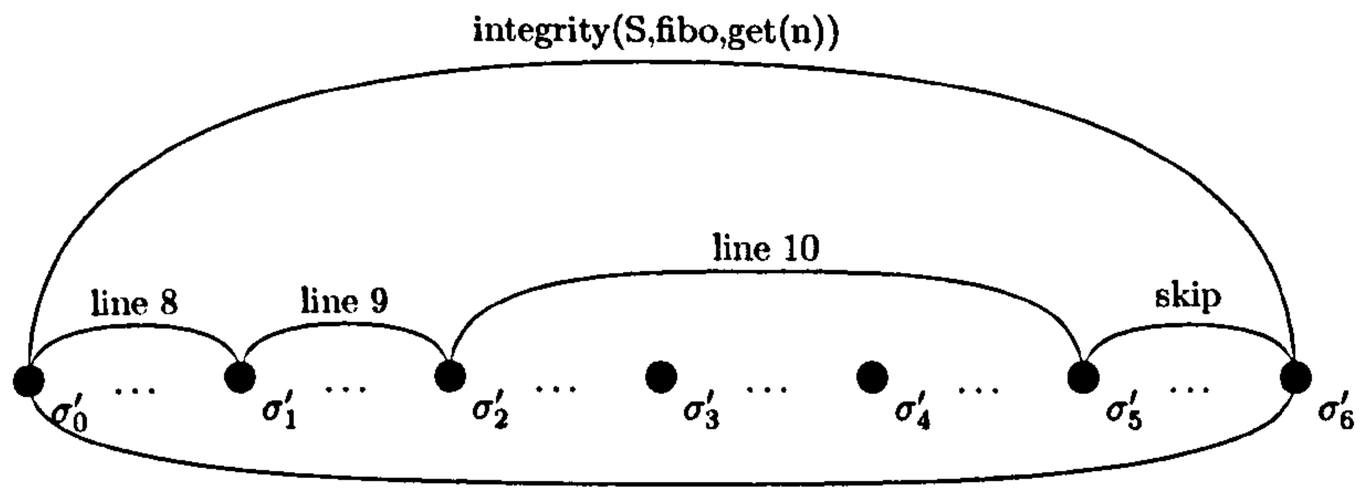


Figure 7.3: 2. Integrity Rule: One previous access to `get`

The last rule then states that whenever `get` has been invoked successfully at least two times before, then the result of the current invocation must be the sum the results of the two previous invocations. Note that this rule checks all possible past time intervals. This is depicted in Figure 7.4.

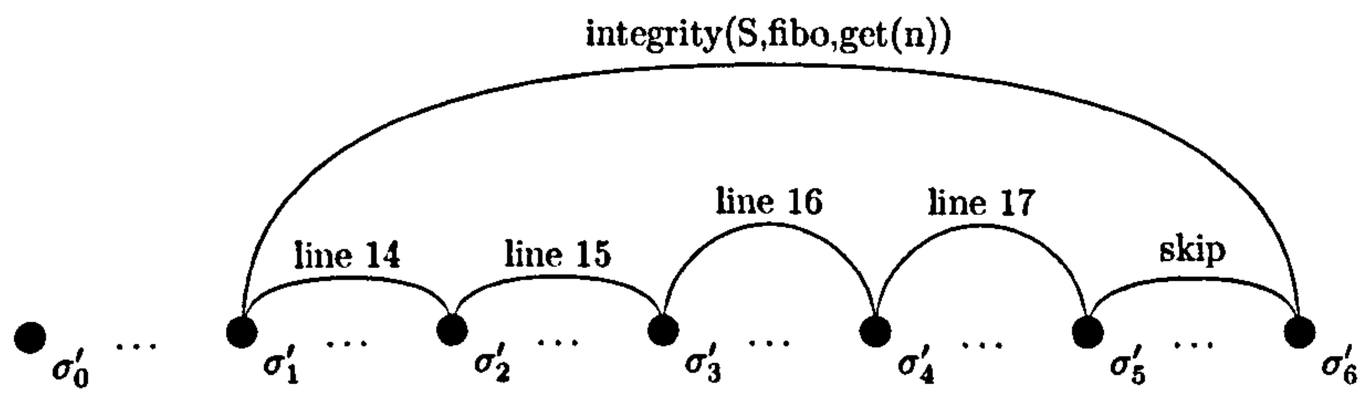


Figure 7.4: 3. Integrity Rule: Two or more previous accesses to `get`

This example shows how integrity policies can be specified on the interface level even if

the constraints are dependent on the past behaviour of the system. A potential application is the verification of interactions with stateful services, to ensure that the service meets constraints that are defined in terms of the integrity policy. In this case we assume that the service instance can be represented by an object.

7.2.6 Simple Policies

Simple policies represent a collection of policy rules that all apply simultaneously. In the SANTA language this is currently represented by grouping a set of rules using parenthesis. Siewe showed in [124] that set theoretic operators can be used to compose simple policies. He also defined some operators to filter rules based on their type (e.g. positive authorisation, negative authorisation rules). These operators for the set theoretic composition of simple policies are currently not included in the SANTA syntax, but can be included without much difficulty.

7.3 Semantics of Policy Rules and Simple Policies

The semantics of policies defines the possible behaviour of boolean ITL state variables, that capture access control decisions, the outcome of integrity checks and the current obligations.

7.3.1 Control Variables

The consequences of policy rules are captured by the boolean state variables. For each type of rule a specific variable, that is subscripted with a subject, object and action, is introduced. Policies define the behaviour of these variables.

Access control Access control decisions are captured by the boolean state variables:

- $autho^{+}(s, o, a) = true$, denotes the positive authorisation of subject s to perform action a on object o.
- $autho^{-}(s, o, a) = true$, denotes the negative authorisation of subject s to perform action a on object o.
- $autho(s, o, a) = true$, denotes the authorisation decision for subject s to perform action a on object o. This decision is enforced by the enforcement mechanisms. If the value is *true* then the corresponding access request succeeds. If the value is *false* it fails.

For environmental policies, viz. policies controlling the access to interfaces of SANTA objects by SANTA agents, the set of all variables capturing access control is:

$$V_{autho,env} = \bigcup_{s \in \mathcal{A}} \bigcup_{o \in \mathcal{O}} \bigcup_{a \in I_o} \{autho^+(s,o,a), autho^-(s,o,a), autho(s,o,a)\}$$

where $\mathcal{A}$ is the set of agents in the SMAS, $\mathcal{O}$ is the set of objects in the SMAS, and I_o is the set of interfaces of the object o. For behavioural policies, viz. policies controlling the sequence of actions an agent can execute, the set is:

$$V_{autho,beh} = \bigcup_{s \in \mathcal{A}} \bigcup_{a \in X_s} \{autho^+(s,s,a), autho^-(s,s,a), autho(s,s,a)\}$$

where $\mathcal{A}$ is the set of agents in the SMAS and X_s the set of actions that are encapsulated in the agent a. The set of all access control related control variables is the union of these two.

$$V_{autho} = V_{autho,env} \cup V_{autho,beh}$$

Obligation Obligations are captured by the boolean state variables $oblig(s,o,a)$. The value *true* denotes the obligation of subject s to perform the action a on object o. The policy model can capture obligations in form of environmental and behavioural policies.

$$V_{oblig,env} = \bigcup_{s \in \mathcal{A}} \bigcup_{o \in \mathcal{O}} \bigcup_{a \in I_o} \{oblig(s,o,a)\}$$

For behavioural policies, viz. policies controlling the sequence of actions an agent can execute, the set is:

$$V_{oblig,beh} = \bigcup_{s \in \mathcal{A}} \bigcup_{a \in X_s} \{oblig(s,s,a)\}$$

The set of all control variables related to obligation is the union of these two.

$$V_{oblig} = V_{oblig,env} \cup V_{oblig,beh}$$

Although obligations can be expressed in environmental policies, SANTA does currently not provide enforcement mechanisms that can enforce them. This means that an obligation of an agent to perform an action can be enforced, but an obligation of an agent to invoke a specific object interface not. The reason for this is that an agent can only execute actions it encapsulates. The definition of an obligation in form of an environmental policy would require the definition of an action by the policy. The enforcement mechanisms would in this case be required to execute the action as it is defined by the policy. This is principally

possible, but leads to concerns whenever the policy is updated, as a policy would then be able to introduce *new* actions in the system. In this case the policy would represent a special form of mobile code. The impact of such policies on the security of the system would require further investigation that is not within the scope of this work.

Integrity Integrity decisions are captured by the boolean state variable $integ(s, o, a)$. The value *true* denotes that the execution of a on o by s satisfies the integrity constraints. For environmental policies the set is:

$$V_{integ,env} = \bigcup_{s \in \mathcal{A}} \bigcup_{o \in \mathcal{O}} \bigcup_{a \in I_o} \{integ(s, o, a)\}$$

For behavioural policies, viz. policies controlling the sequence of actions an agent can execute, the set is:

$$V_{integ,beh} = \bigcup_{s \in \mathcal{A}} \bigcup_{a \in X_s} \{integ(s, s, a)\}$$

The set of all control variables related to integrity is the union of these two.

$$V_{integ} = V_{integ,env} \cup V_{integ,beh}$$

7.3.2 Policy Scope and Free Variables

In the policy specification the set variables *Subj*, *Obj* and *Act* are *free* variables that denote the set of subjects, set of objects and the set of actions in the scope of the policy. They are free in the policy specification and are bound in the semantics of the $SMAS_3$ (see Chapter 8). These free set variables can be referenced in the syntax of the policy using the keywords **subjects**, **objects** and **actions**.

$$[\![\textbf{subjects}]\!] \mathrel{\widehat{=}} Subj \quad [\![\textbf{objects}]\!] \mathrel{\widehat{=}} Obj \quad [\![\textbf{actions}]\!] \mathrel{\widehat{=}} Act$$

Similarly the variables s, o and a are free variables in the specification of policy rules. They denote any subject, object or action within the policy scope. They are bound by the semantics of a simple policy (see Section 7.3.4). They can be referenced in the syntax of rules using the keywords **S**, **O** and **A**.

$$[\![\textbf{S}]\!] \mathrel{\widehat{=}} s \quad [\![\textbf{O}]\!] \mathrel{\widehat{=}} o \quad [\![\textbf{A}]\!] \mathrel{\widehat{=}} a$$

The following example illustrates:

```
allow (S, file, read) when S = bob or S = alice
```

Assume that the free set variables are bound by the system semantics as follows:

$$Subj = \{\texttt{bob}, \texttt{alice}, \texttt{eve}\} \quad Obj = \{\texttt{file}\} \quad Act = \{\texttt{read}, \texttt{write}\}$$

then the semantics of the rule would bind the variable s as follows:

$$\forall_{s \in Subj} [\![\textbf{allow (S, file, read) when S=bob or S=alice}]\!]$$

where in the SANTA syntax s is represented by the keyword **S**. This means that the rule that references the free variable **S** is instantiated to mean the conjunction of:

$$[\![\textbf{allow (bob, file, read) when bob=bob or bob=alice}]\!] \wedge$$
$$[\![\textbf{allow (alice, file, read) when alice=bob or alice=alice}]\!] \wedge$$
$$[\![\textbf{allow (eve, file, read) when eve=bob or eve=alice}]\!]$$

The use of the free variables has the advantage that it defines rules more abstract and allows different instantiations of the same rule, dependent on the binding of the variables in the simple policy. It also provides a way to express requirements that apply to several individuals more compactly — especially if the premise is more complicated than in the above example.

7.3.3 Rules

Policy rules define the behaviour of the access control, obligation and integrity control variables. The semantic of a rule is captured by the operator *always followed by* that has been introduced by Siewe in [124]. The operator captures the dependency of the policy decision on the preceding behaviour as follows:

$$f \mapsto w \mathrel{\widehat{=}} \boxdot ((\Diamond f) \supset \mathsf{fin}(w)) \tag{7.1}$$

The intuition of the operator is that whenever f holds in the left neighbourhood of a state it implies that the state formula w holds in that state. This is depicted in Figure 7.5.

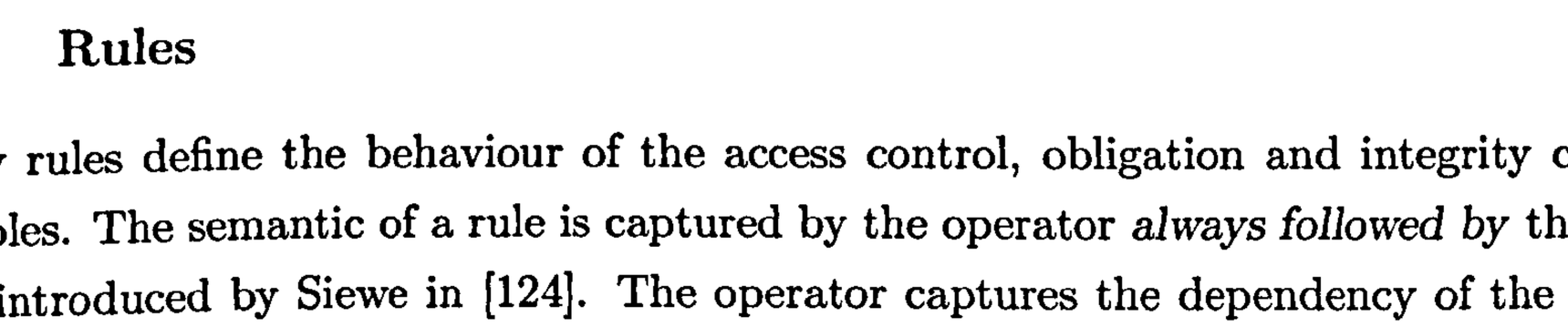


Figure 7.5: Operator Always-Followed-By

The implication $\Diamond f \supset \mathsf{fin}\ w$ means that w can be *true* in a state even if f does not hold in the left neighbourhood of that state. In Figure 7.5 this is the case for the state σ'_4. This is addressed by transforming the policy into a *complete* policy, that defines the access decision in every state. This is important for the enforcement of policies and discussed in detail in Chapter 8.

Premise

The syntax that is used in the premise is actually an ASCII representation of a subset of ITL formulae. The semantics of the constructs is straight forward.

$$[\![\texttt{f ; g}]\!] \mathrel{\widehat{=}} [\![\texttt{f}]\!] \,;\, [\![\texttt{g}]\!] \tag{7.2}$$

$$[\![\texttt{f and g}]\!] \mathrel{\widehat{=}} [\![\texttt{f}]\!] \wedge [\![\texttt{g}]\!] \tag{7.3}$$

$$[\![\texttt{f or g}]\!] \mathrel{\widehat{=}} [\![\texttt{f}]\!] \vee [\![\texttt{g}]\!] \tag{7.4}$$

$$[\![\textsf{sometime f}]\!] \mathrel{\widehat{=}} \Diamond [\![\texttt{f}]\!] \tag{7.5}$$

$$[\![\textsf{always f}]\!] \mathrel{\widehat{=}} \Box [\![\texttt{f}]\!] \tag{7.6}$$

$$[\![\textsf{keep f}]\!] \mathrel{\widehat{=}} \mathsf{keep}\,[\![\texttt{f}]\!] \tag{7.7}$$

$$[\![\textsf{if expr then f else g}]\!] \mathrel{\widehat{=}} (\texttt{expr} \wedge [\![\texttt{f}]\!]) \vee (\neg\,\texttt{expr} \wedge [\![\texttt{g}]\!]) \tag{7.8}$$

$$[\![\textsf{exists i < expr : f}]\!] \mathrel{\widehat{=}} \exists \texttt{i} \bullet \texttt{i} > 0 \wedge \texttt{i} \leq \texttt{expr} \wedge [\![\texttt{f}]\!] \tag{7.9}$$

$$[\![\texttt{expr : f}]\!] \mathrel{\widehat{=}} \mathsf{len}(\texttt{expr}) \wedge [\![\texttt{f}]\!] \tag{7.10}$$

$$[\![\texttt{list : f}]\!] \mathrel{\widehat{=}} [\![\textsf{exists e in list : (e: f)}]\!] \tag{7.11}$$

The existential and universal quantification over lists (viz. **exists e in expr : f**) is an abbreviation that allows to directly reference the list elements. Its semantics can be easily expressed in terms of the bounded quantification.

Basic Rule Definitions

The consequence of a rule determines the type of the rule and the subjects, objects and actions the rule applies to. The operator *always-followed-by* is used to capture the relation between the premise of a rule and its consequence. The basic definition of the SANTA rule constructs is then as follows:

$$[\![\textsf{allow (S,O,A)when premise}]\!] \mathrel{\widehat{=}} [\![\texttt{premise}]\!] \;\mapsto\; \mathit{autho}^{+}(s, o, a) \tag{7.12}$$

where **S**, **O** and **A**, denote the free variables s, o and a in the rule. These can also occur in the premise. Instead of the free variables, identifiers of subjects, objects and actions can be used as constants. This represents an instantiation of the variables (e.g. $s = const$). The definition for the other types of rules is similar:

$$[\![\textbf{deny (S,O,A)when premise}]\!] \mathrel{\widehat{=}} [\![\texttt{premise}]\!] \mapsto autho^{-}(s,o,a) \tag{7.13}$$

$$[\![\textbf{decide (S,O,A)when premise}]\!] \mathrel{\widehat{=}} [\![\texttt{premise}]\!] \mapsto autho(s,o,a) \tag{7.14}$$

$$[\![\textbf{oblige (S,O,A)when premise}]\!] \mathrel{\widehat{=}} [\![\texttt{premise}]\!] \mapsto oblig(s,o,a) \tag{7.15}$$

$$[\![\textbf{integrity (S,O,A)when premise}]\!] \mathrel{\widehat{=}} [\![\texttt{premise}]\!] ; \textbf{skip} \mapsto integ(s,o,a) \tag{7.16}$$

For example the semantics of the policy rule:

```
allow (S, file, read) when owner(S, file)
```

is then

$$owner(s, \texttt{file}) \mapsto autho^{+}(s, \texttt{file}, \texttt{read})$$

where s is a free variable and we assume that the predicate `owner` is defined by the policy or the enforcement mechanism.

The definitions all rules follows the same pattern, except for integrity policies. Here the skip in the semantics of the rule guarantees that the premise as it is defined in the rule does not reference the state in which the decision is made. This is necessary, because the enforcement of integrity rules states that a rule may only have succeeded in a state if its integrity constraints are met. The dependency of the successful execution on the integrity prohibits that the integrity depends itself on the successful execution in the same state, as this could otherwise lead to an infinite recursive definition of an integrity rule that would not be enforceable.

In the following the definitions of authorisation (Equations (7.12), (7.13) (7.14)) and integrity rules (Equation (7.16)) are augmented to allow for the referencing of parameters, temporary results and the enforcement time in their premise. Obligation rules are currently only allowed in behavioural policies and can therefore not contain references to parameters (SANTA actions are not parametrised).

Referencing Results in Integrity Rules

Integrity rules can reference temporary results of an action/interface execution, using the prime notation. The primed variables have the same values that the agent/object variables would be assigned if the action/interface execution succeeded. This allows to write assertions on the action/interface execution to determine the success or failure of the action/interface based on the results.

Auxiliary Variables The temporary results have been modelled in Chapters 5 and 6 using local variables. The scope of these variables is limited to the execution of the action/interface and cannot be observed otherwise. For the *enforcement* of policies this is without consequence, as the refinement of policies to concrete enforcement mechanisms affects only those parts of the system that are within the scope of the local variable definition (see Chapter 9). However, for the abstract specification of policies the problem of accessibility persists. The following shows how local variables that capture the temporary results of the computation can be replaced by dedicated auxiliary variables at the agent/object level.

The set of local variables that are created during execution of an action/interface is determined by the parameters (for interfaces) and the set of agent/object variables that are modified by the action/interface. For example in the following agent specification:

```
1 agent ag :
2   var x = 0
3   when true do computex : { /* some computation on x */ }
4   deliberation : external
5 end
```

The execution of the action `computex` creates exactly one local variable x' to store the temporary result of the agent variable `x`. Instead of using the local variable, the set of object variables is augmented with the auxiliary variable `computex.x`. The naming convention for these auxiliary variables is to precede the agent/object variable with the respective action/interface name.

The auxiliary variable `computex.x` can be used instead of the local variable x' in the execution of the action because the semantics of the SMAS guarantees that the execution of an agent's actions/object's interfaces are mutually exclusive, viz. there is no concurrent modification.

Another concern is the validity of the values of the auxiliary variables, i.e. that they are only accessible in the right context. The auxiliary variables for a specific action/interface are initialised at the beginning of the action/interface execution. The actual computation is then performed on the auxiliary variables. If the action/interface execution succeeds, the values of the auxiliary variables are copied to the corresponding agent/object variables (*successful execution*) or left unchanged (*failed execution*). The values of the auxiliary variables are maintained until the next execution of the same action/interface. This means that the auxiliary variables always contain the result of the last execution of the action/interface. In particular, when an integrity decision has to be made for an action/interface, the auxiliary variables contain the results of the computation of that action/interface and the decision on the success of failure can be based on their values. This ensures that the

values in the auxiliary variables contain updated values at the time the check is made.

In the following it is assumed that for every agent/object variable, that is modified by the action/interface execution, an auxiliary variable with the same name as the agent/object variable, preceded by the name of the action/interface is introduced as a substitution of the local variables that have been introduced in Chapters 5 and 6.

Augmented Semantics of Integrity Rules The semantics of integrity rules is augmented to define the meaning of the primed variables as follows:

$$\begin{aligned} &[\![\,\textsf{integrity}\ \ (\textsf{S},\textsf{O},\textsf{A})\,\textsf{when}\ \texttt{premise}\,]\!] \mathrel{\widehat{=}} \\ &(\exists \overline{v'} \bullet (\overline{v'} = \textsf{fin}\ z.\overline{v}) \wedge ([\![\texttt{premise}]\!]\,;\textsf{skip})) \ \mapsto\ integ(s,o,a) \end{aligned} \tag{7.17}$$

Again **S**, **O** and **A** denote the free variables s, o and a in the rule. Here $\overline{v'}$ is a list of local static variables. Each variable $v' \in \overline{v'}$ represents the value of the corresponding auxiliary variable $x.v$ in the final state of the interval over which the rule is evaluated. This means in the same state in which the action succeeded or failed. Any primed agent/object variable references the corresponding local static variable v' of the rule. The following simple example illustrates:

```
integrity(ag,ag,computex) when 0: x' > x
```

The rule asserts, that the new value that has been computed for `x` (represented by x') is greater than the value of x before the execution started. A part of the behaviour defined by the action `computex` and the above integrity rule are depicted in Figure 7.6 below.

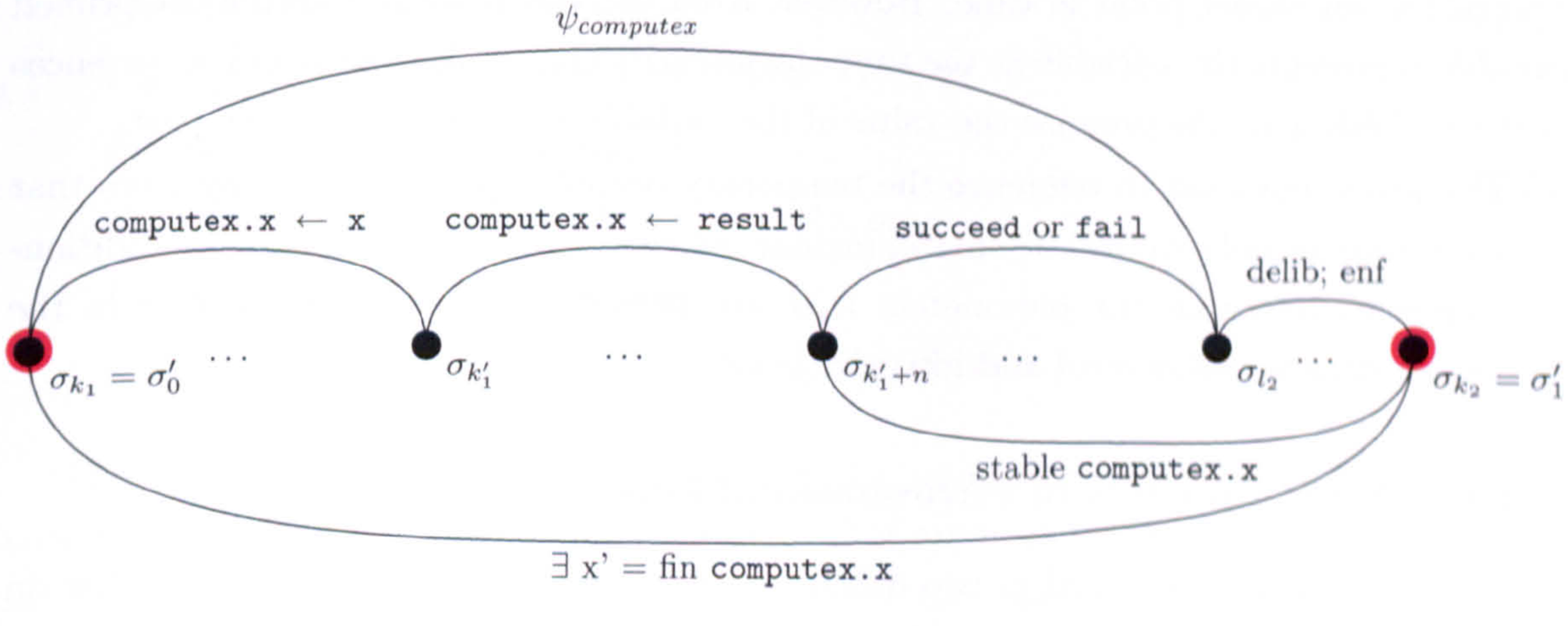


Figure 7.6: Primed Variables in Integrity Rules

The action `computex` defines the behaviour of the agent over the interval $\sigma_{k_1} \ldots \sigma_{l_2}$. The auxiliary variable `computex.x` is initialised to the agent variable `x` in state $\sigma_{k'_1}$. The result

of the action execution is available in the auxiliary variable after n steps. From thereon the auxiliary variable remains stable until the execution of the next action in state σ_{k_2}. In state $\sigma_{k'_1+n}$ the agent will either succeed in the execution or fail.

The definition of the policy is more abstract than the definition of the system. For a vigilant agent the states that are selected for the policy interpretation (σ'_0 and σ'_1) are those where the agent is *ready*[3]. In Figure 7.6 these are the states σ_{k_1} and σ_{k_2} (highlighted in red). The semantics of the above rule over the interval $\sigma' = \sigma'_0\sigma'_1$ is:

$$(\exists \mathtt{x'} \bullet (\mathtt{x'} = \mathsf{fin}\ \mathtt{computex.x}) \wedge ((\mathtt{x'} > \mathtt{x} \wedge \mathsf{len} = 0)\ ;\ \mathsf{skip})) \mapsto \mathit{integ}(\mathtt{ag}, \mathtt{ag}, \mathtt{computex})$$

This states that there is a local static variable x' that has the same value over the interval $\sigma'_0\sigma'_1$ as the auxiliary variable computex.x has in state σ'_1. The rule defines that *integ*(ag, ag, computex) is *true* in state σ'_1 if the value of x' is greater than the value of x in state σ_0.

Informally the enforcement of an integrity rule means that an action must not succeed if its integrity constraints are not met. Given that the value of the auxiliary variable computex.x does not change in the interval $\sigma_{k'_1+n} \ldots \sigma_{k_2}$ the value of x' can be determined earlier, i.e. in state $\sigma_{k'_1+n}$. By refining the non-deterministic choice between the action success and failure into a conditional choice that depends on the constraints defined in the integrity rule, the integrity rule can be enforced to guarantee the action does not succeed if the constraint is not met.

From the viewpoint of the premise, the primed variables are *prophecy variables*. This means they reference the result of the computation (only available in the final state of the interval) at an earlier point in time. However, from the viewpoint of the rule the primed variable represents the variable at the time the integrity check takes place and occurrences of the variable x in the premise the value of the variable at some point in the past.

The prime notation to reference the temporary results applies for integrity rules that are expressed as behavioural or environmental policies. Environmental policies additionally allow to reference the parameters that are passed in interface invocations in the specification of access control and integrity rules.

Referencing Parameters in Environmental Policies

Parameters can be referenced in two different ways. The most common usage is that an authorisation or integrity rule depends on the parameters that are being passed. Alternatively parameters can be referenced in the premise of rules using the done(x,y,z(p0,...,pn)) construct.

[3]The enforcement is discussed in detail in Chapter 8

For example a delegation rule is an authorisation that controls a specific, parameterised interface and is almost always dependent on the parameters that are passed. The delegation rule that allows all subjects to delegate their own rights is a good illustration:

```
1 allow (S,O,delegate(g1,d1,y1,z1)) when 0: S = g1
```

This rule denotes, that the first parameter $g1$, identifying the delegator of the right must match the identity of the subject `S` that is invoking the interface. The values of the other parameters are not used in this rule.

Similarly to the case of temporary results, input and output parameters can be represented by auxiliary variables. We assume here that the enforcement of environmental policies ensures that all parameters have been appropriately set before the evaluation of the policy rule. The naming convention for parameter auxiliary variables is to prefix the name of the parameter with the interface name. Assuming that the interface `delegate` is defined as in Listing 7.5, the following four auxiliary variables for the parameters are introduced: `delegate.g`, `delegate.d`, `delegate.y`, and `delegate.z`.

The semantics of rules that reference parameters is augmented along the same lines as discussed for the referencing of temporary results.

$$[\![\texttt{allow (S,O,A}(\overline{p})\texttt{) when premise}]\!] \mathrel{\widehat{=}} (\exists \overline{p} \bullet (\overline{p} = \mathsf{fin}\ z.\overline{p}) \wedge [\![\texttt{premise}]\!]) \mapsto autho^{+}(s,o,a) \tag{7.18}$$

$$[\![\texttt{deny (S,O,A}(\overline{p})\texttt{) when premise}]\!] \mathrel{\widehat{=}} (\exists \overline{p} \bullet (\overline{p} = \mathsf{fin}\ z.\overline{p}) \wedge [\![\texttt{premise}]\!]) \mapsto autho^{-}(s,o,a) \tag{7.19}$$

$$[\![\texttt{decide (S,O,A}(\overline{p})\texttt{) when premise}]\!] \mathrel{\widehat{=}} (\exists \overline{p} \bullet (\overline{p} = \mathsf{fin}\ z.\overline{p}) \wedge [\![\texttt{premise}]\!]) \mapsto autho(s,o,a) \tag{7.20}$$

$$[\![\texttt{integrity (S,O,A}(\overline{p})\texttt{) when premise}]\!] \mathrel{\widehat{=}} (\exists \overline{p}, \overline{v'} \bullet (\overline{p} = \mathsf{fin}\ z.\overline{p}) \wedge (\overline{v'} = \mathsf{fin}\ z.\overline{v}) \wedge ([\![\texttt{premise}]\!] ; \mathsf{skip})) \mapsto integ(s,o,a) \tag{7.21}$$

Here the local static variables that capture the parameter values $\overline{p}$ are assigned to the corresponding values of the auxiliary variables that are used for the parameters $z.\overline{p}$ in the final state. This means that parameters specified in the consequence of rules reference the value of the auxiliary parameter variables at the time of the decision. It is easy to see that without parameters the augmented rule definitions are identical to those previously introduced in Equations (7.12) to (7.15) and (7.17). It is also possible to reference parameters in the premise of rules at a different point in time.

Referencing Parameters of Past Interface Executions To reference a parameter at a different point in time requires the exact specification of that time point. To ensure that the parameter values that are stored in the auxiliary variables are *fresh*, the time point is determined using the construct **done(x,y,z($\overline{q}$))** in the premise of a rule. $\overline{q}$ is a list of local static variables, that have the same value as the parameters at the time where **done(x,y,z)** is *true*. This guarantees that the parameters $\overline{q}$ are referenced in a state, where the action has successfully executed, viz. the auxiliary variables contain meaningful values. The concrete state depends on where in the specification of the premise the construct is used.

The mechanisms used to capture this is similar to the parameter definitions in the consequence of rules. For each referenced parameter a local static variable is introduced that captures the value of the parameter at the specified point in time.

Contrary to the prime variables and the parameters in the consequence of the rules, the values that are assigned to these local static variables are not the value of the auxiliary variable in the final state, but the value in the state where the control variable referenced by **done(x,y,z)** is true.

For every occurrence of the construct **done(x,y,z($\overline{q}$))**, where $\overline{q}$ denotes the parameter list of the interface z the set $\overline{q}$ is introduced as local static variables in the premise of the rule. The corresponding construct **done(x,y,z($\overline{q}$))** is replaced with the expression **done(x,y,z)** $\wedge\ \overline{q} = z.\overline{q}$, viz. the assignment of the local static variables $\overline{q}$ to the values of the corresponding auxiliary variables for the parameters in the same state. This is a purely syntactic transformation, that allows to reference parameters with relative ease.

The semantics of policy rules that was provided in Equations (7.18) to (7.21) is augmented to allow for the referencing of past parameter values as follows:

$$[\![\textbf{allow (S,O,A}(\overline{p}))\ \textbf{when premise}]\!] \mathrel{\widehat{=}} (\exists \overline{p}, \overline{q} \bullet (\overline{p} = \mathsf{fin}\ z.\overline{p}) \wedge [\![\texttt{premise}']\!]) \mapsto autho^{+}(a, o, a) \tag{7.22}$$

$$[\![\textbf{deny (S,O,A}(\overline{p}))\ \textbf{when premise}]\!] \mathrel{\widehat{=}} (\exists \overline{p}, \overline{q} \bullet (\overline{p} = \mathsf{fin}\ z.\overline{p}) \wedge [\![\texttt{premise}']\!]) \mapsto autho^{-}(s, o, a) \tag{7.23}$$

$$[\![\textbf{decide (S,O,A}(\overline{p}))\ \textbf{when premise}]\!] \mathrel{\widehat{=}} (\exists \overline{p}, \overline{q} \bullet (\overline{p} = \mathsf{fin}\ z.\overline{p}) \wedge [\![\texttt{premise}']\!]) \mapsto autho(s, o, a) \tag{7.24}$$

$$[\![\textbf{integrity (S,O,A}(\overline{p}))\ \textbf{when premise}]\!] \mathrel{\widehat{=}} (\exists \overline{p}, \overline{q}, \overline{v'} \bullet (\overline{p} = \mathsf{fin}\ z.\overline{p}) \wedge (\overline{v'} = \mathsf{fin}\ z.\overline{v}) \wedge ([\![\texttt{premise}']\!]\ ;\ \mathsf{skip})) \mapsto integ(s, o, a) \tag{7.25}$$

Here $\overline{q}$ is a list of all local static variable identifiers that are used in the premise to

reference parameters in the constructs done(x,y,z($q_1, \ldots, q_n$)). These can be obtained by simple syntactic analysis of the rule's premise. premise$'$ denotes the original premise, where every occurrence of the construct

```
done(x,y,z(q1,...,qn))
```

is rewritten as:

```
done(x,y,z) and q1 = z.q1 and ... and qn = z.qn
```

with z.q_n denoting the auxiliary variable of the corresponding parameter.

The choice to reference parameters in this form and rewrite the original specification of the premise guarantees that auxiliary variables for parameters can only be accessed at states where they contain meaningful values. The following example of a rule illustrates.

Any subject is denied to delegate a right to a delegatee, if it was the case that the subjects delegated the same right previously to another delegatee

```
1  deny (S,O,delegate(g1,d1,y1,z1)) when
2      sometime done(S,O,delegate(g1,d2,y1,z1)) and
3      S = g1 and d1 <> d2
```

Line 1 defines the negative authorisation for any subject to delegate the right to perform action z1 on object y1 from the delegator g1 to the delegatee d1. The premise of the rule states in Line 2 that the rule fires if the subject did previously delegate the same right from the same delegator g1 to the another delegatee d2 provided that (Line 3) the delegator was in both cases the subject itself and the delegatee of the current request is different to the one of the previous delegation. The use of the same name for different parameters is permissible and denotes that both parameters values must be equal. The semantics of the above rule is then:

$$
\begin{aligned}
&(\exists g1, d1, y1, z1, d2 \cdot (g1 = \text{fin } \texttt{delegate.g}) \wedge (d1 = \text{fin } \texttt{delegate.d}) \wedge \\
&\quad (y1 = \text{fin } \texttt{delegate.y}) \wedge (z1 = \text{fin } \texttt{delegate.z}) \wedge \\
&\quad (\Diamond(done_{o,s,\texttt{delegate}} \wedge (g1 = \texttt{delegate.g}) \wedge (d2 = \texttt{delegate.d}) \wedge \\
&\qquad (y1 = \texttt{delegate.y}) \wedge (z1 = \texttt{delegate.z})) \\
&\quad) \wedge (s = g1) \wedge (d1 \neq d2)) \\
&\quad \mapsto \; autho^{-}(s, o, \texttt{delegate})
\end{aligned}
$$

This is the fully expanded semantics of the rule. The keywords S, O are represented by the variables s and o that range over the set of subjects $Subj$ and objects Obj. By syntactic analysis of the rule we can establish that five distinct variables are used to referencing

parameters (this is to avoid the need for an explicit quantification of these variables). The variables are **g1**, **d1**, **d2**, **y1** and **z1**. They are bound by the existential quantification in the premise of the rule. The variables **g1**, **d1**, **y1** and **z1** reference parameters in the consequence of the rule and are therefore equal to the values of their corresponding auxiliary variables in the final state.

The parameters **g1**, **d2**, **y1** and **z1** are referenced in the premise of the rule using the construct **done(S,O,delegate(g1,d2,y1,z1))**. Listing the parameters in the **done** construct is an abbreviation for:

$$\textbf{done(S,O,delegate)} \wedge \textbf{g1} = \textbf{delegate.g} \wedge \textbf{d2} = \textbf{delegate.d} \wedge$$
$$\textbf{y1} = \textbf{delegate.y} \wedge \textbf{z1} = \textbf{delegate.z}$$

done(S,O,delegate) references the boolean control variable $done_{O,S,delegate}$ that indicates the successful execution. This means that the action has successfully executed in that state and the auxiliary variables for the parameters are at that state equal to the static variables **g1**, **d2**, **y1** and **z1**. The last two conjuncts represent the constraints that the delegator must be the subject itself and that the delegatee must be different. The semantics of the rule is depicted in Figure 7.7.

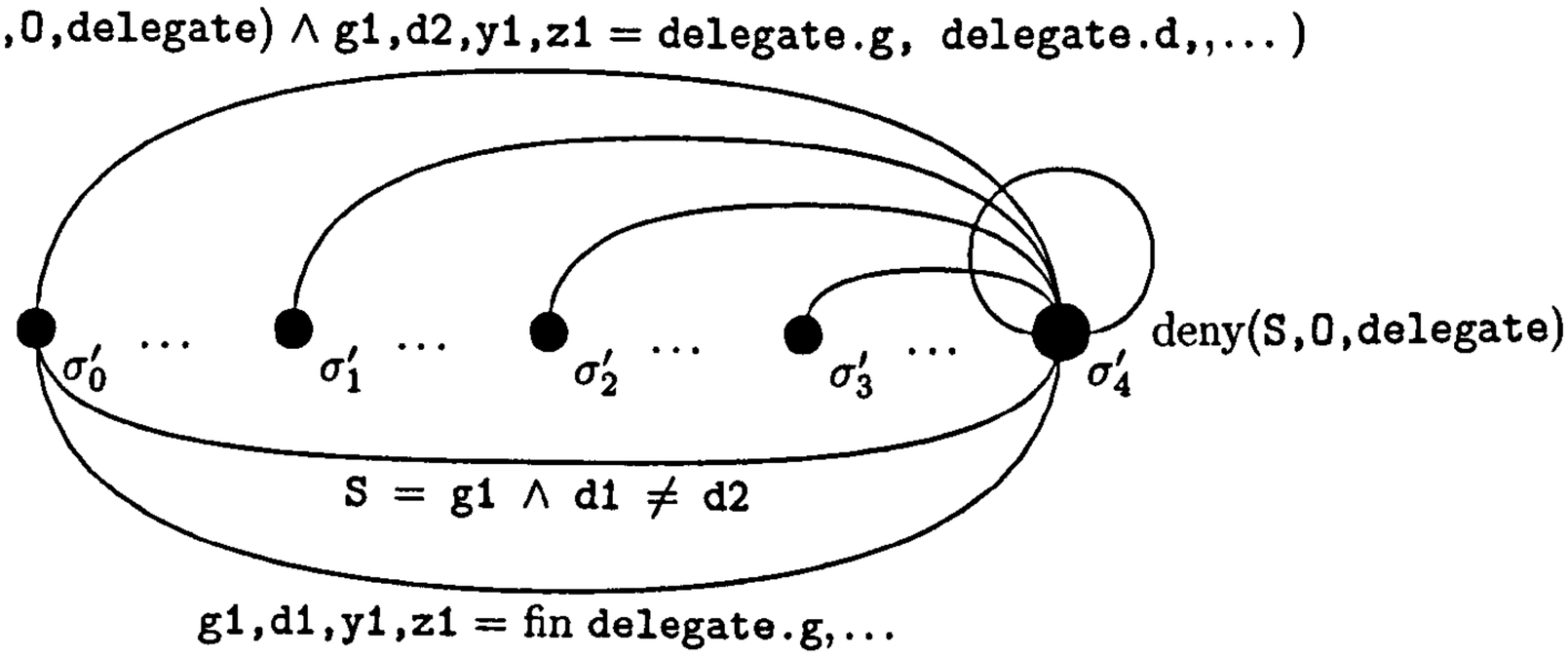


Figure 7.7: Example of referencing parameter values in the past

Given that the decision is derived for σ'_4 then the rule states the there exists a set of local static variables **g1**, **d1**, **d2**, **y1** and **z1**, for which **g1**, **d1**, **y1** and **z1** equal the value of the auxiliary variables in the final state (bottom interval). The **sometime** states that there is some suffix interval (the intervals depicted above the state sequence) for which the control variable denoted by **done(S,O,delegate)** is true and where the values of the local static variables **g1**, **d2**, **y1** and **z1** equal the values of the auxiliary variables. The overall interval must also satisfy that $S = \textbf{g1} \wedge \textbf{d1} \neq \textbf{d2}$ for the rule to fire.

All the local static variables are used as prophecy variables. Their value is defined at

some point in the interval that is described by the premise. In the example this is the final state and some state in which the delegation has been successfully performed before. Static variables do not change their value over the interval, thus defining their value in any state of the interval suffices. Therefore the comparison $d1 \neq d2$ can be made in the initial state of the interval provided that the whole interval is known. This is always the case for the premise of a rule as it references the past.

As discussed earlier for some requirements it is necessary to reference the abstract time that elapsed since the enforcement of the rule started.

Referencing Enforcement Time The keyword `T` is used to reference the abstract time that elapsed between the start of the rule enforcement and the decision. `T` can be used as an expression and yields the value of a policy specific clock in the final state of the premise. The behaviour of the clock variable *Clock* is defined in the semantics of simple policies (see Section 7.3.4). Similar to the augmentation of the rule definitions before, each premise is augmented with a local static variable `T`, that is initialised to the value *Clock* in the final state. The definition of the augmented rule semantics is omitted, as the extension is along the same lines as the extensions for temporary results and parameter values.

7.3.4 Simple Policies

A simple policy is a collection of rules that apply simultaneously. The semantics of the simple policy binds the variables s, o and a that are free in the rules using the universal quantification over the sets variables *Subj*, *Obj* and *Act*. Additionally a clock variable that is local to the simple policy is introduced. This clock provides the abstract time that elapsed since the rule started to be enforced. The definition is given below:

$$\begin{aligned}[p] \mathrel{\widehat{=}} & \exists Clock \bullet ((Clock = 0) \wedge (Clock \text{ gets } Clock + 1) \wedge \\ & \forall s \in Subj \bullet \forall o \in Obj \bullet \forall a \in Act \bullet \bigwedge_{r \in p} r)\end{aligned} \tag{7.26}$$

Where the variables s, o and a can occur freely in r. The value of the clock in the state in which a rule decision is made is accessible using the keyword `T`.

7.4 Policy Composition

The concept of composition is powerful both for specification and validation purposes. In the specification, it helps the policy administrator to focus on a specific set of requirements, that does apply in a specific situation, or to a subset of the entities in the system. This eases

especially the comprehension of existing policies, which is a key factor in the maintenance of policies.

The advantage for the validation and verification of policies lies in the fact the semantics of each policy component and the semantics of each composition is formally defined. It is possible to proof properties of small and simple policies and infer from those the properties of their compositions. For example, to prove that the sequential composition of two policies does never allow a specific access can be decomposed in the proof of its parts, viz. it is sufficient to prove that neither the first nor the second policy does allow the access.

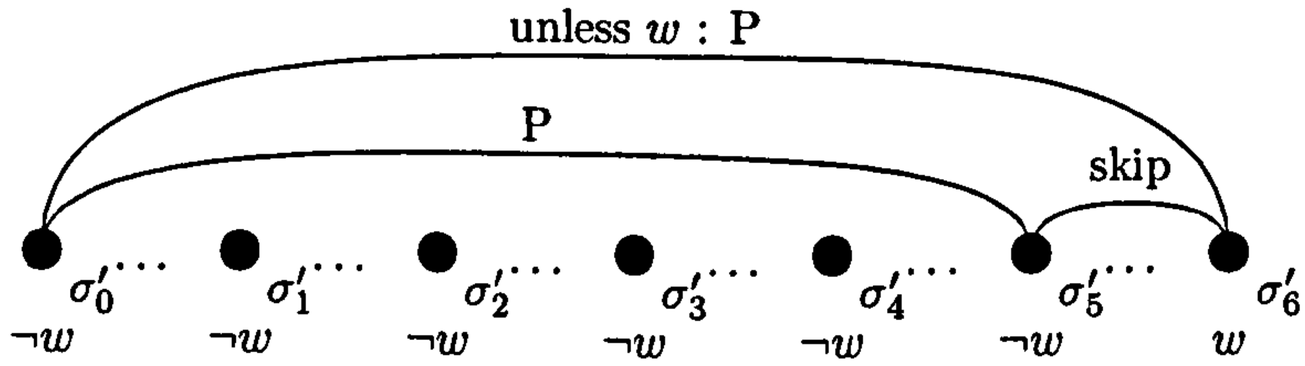
Two forms of composition can be distinguished: a) the composition along the temporal axis and b) the composition along the structural axis. These are detailed in the following.

7.4.1 Temporal Composition

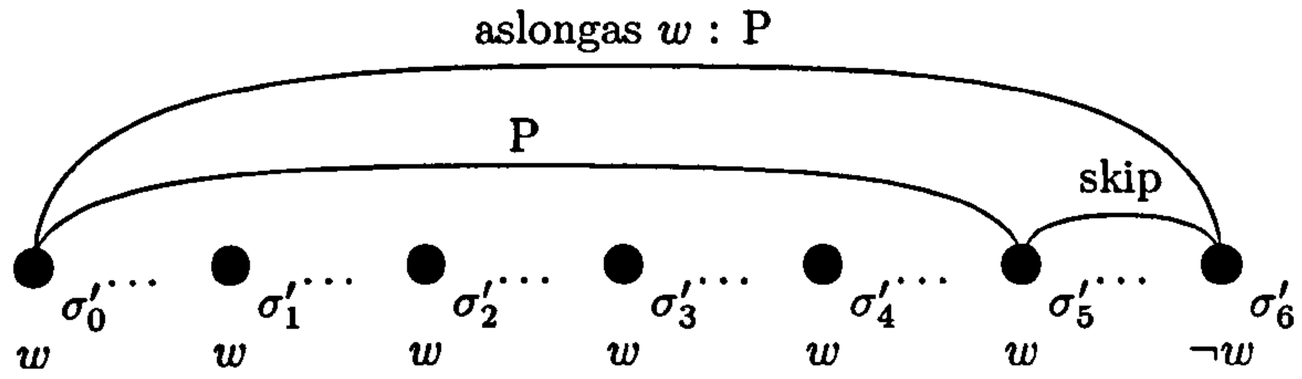
Temporal composition of policies has been introduced by Siewe in [124]. The composition operators are here presented shortly for the completeness of the policy language description. The temporal composition of policies yields policies that dynamically change over time and on events. It allows for the independent specification of policies that apply only in a certain situation. The composition operators are then used to define the conditions of the policy change.

Sequence of Policies

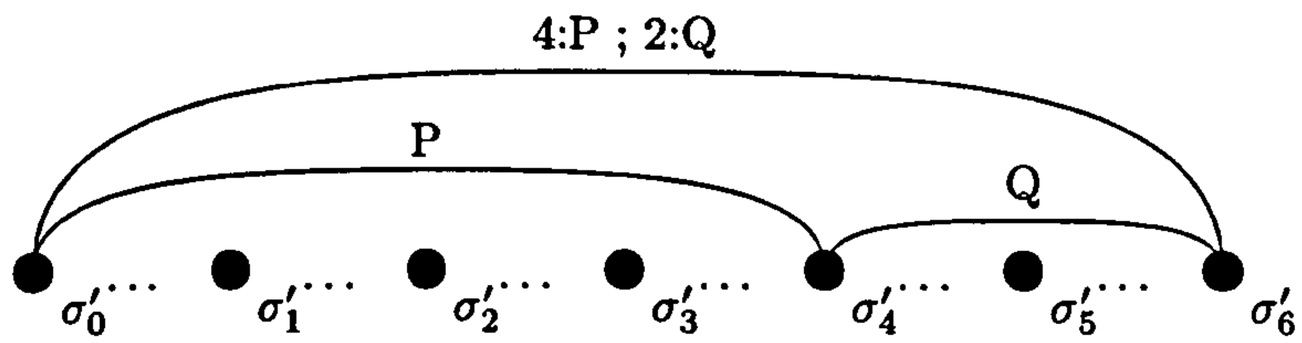
Two policies can be sequentially composed. This is denoted by the semicolon operator (;). The composition `P ; Q` of two policies `P` and `Q` means that first `P` is enforced and then `Q`. However, for the sequential composition to be enforceable it is necessary to define the exact condition for the transition from the first policy to the second. This can be achieved using the operators **unless**, **aslongas** or the explicit timing using the operator `t : P`. The informal meaning of these operators is depicted below:



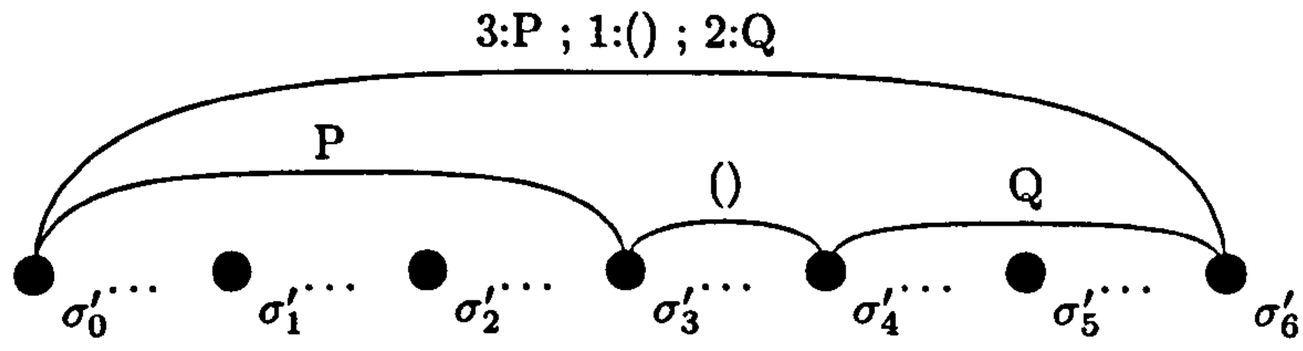
The policy `P` holds *unless* the condition w becomes true. The definition of the operator ensures that `P` holds only over the states where w does not hold, viz. from the first to the penultimate state of the interval. The policy that is enforced in the final state must be defined by a subsequent policy.



The policy P holds *as long as* the condition w is true. The definition of the operator ensures that P holds only over the states where w does hold. The time over which the policy is enforced can also be explicitly specified.



In the depicted case the policy P is enforced over the first four states and the policy Q over the next two states. In the shared state σ'_4 both policies are applies. If this is not desired, a unit interval in which the *empty* policy (denoted by the empty simple policy ()) is enforced must be inserted between both policies. This is depicted below:



A good example where the sequential composition of policies provides an advantage over other approaches is the policy that applies to the operational procedures at airports. It is easy to separate the overall requirements into those that apply under normal circumstances and those that apply when a security warning has been issued. We assume here that the system that is supporting all activities at the airport provides a means for authorised individuals to raise a security alert. The sequential composition of policies in SANTA captures this natural separation during the specification of the policy. We assume that the policies for both situations can be specified individually and are captured in the policies `norm_policy` and `alert_policy`.

```
1  policy norm_policy : /* ... */  end
2  policy alert_policy : /* ... */  end
3
4  policy composition :
5    (unless done(S, system, alert) : norm_policy) ; alert_policy
6  end
```

We assume the definitions of both policies `norm_policy` and `alert_policy`. The composition of both is defined in the policy `composition`, which states that *unless* an alert has been issued, the policy `norm_policy` applies and thereafter the policy `alert_policy`. This example shows the transition between exactly two policies.

Conditional Policy

Consider the case that the action `alert` defines a single input parameter `in level` and sets the new security level in the system to its value. In this case a decision must be made based on the passed parameter to determine which policy is to be enforced. This could be encoded using the conditional choice as follows:

```
1  policy norm_policy : /* ... */  end
2  policy alert_policy : /* ... */  end
3
4  policy composition :
5    (unless done(S, system, alert(level)) : norm_policy) ;
6    if (done(S, system, alert(level)) then (
7            if (level = norm_level) then norm_policy
8       else if (level = high_level) then alert_policy
9    )
10 end
```

The policy enforcement starts with the policy `norm_policy` until the action `alert` has been executed. The policy then changes into a conditional policy that enforces either the policy `norm_policy` or the policy `alert_policy`, dependent on the passed parameter. It is easy to see how the policy can be extended to distinguish multiple security levels. However, the example allows only for exactly one transition, viz. after the level has been set once the policy does not change anymore.

Iteration of Policies

The intuition behind the example is obviously that the policy changes every time the security level of the system is changed. To express this, the iteration construct **repeat** is used.

```
1  policy composition :
2    (unless done(S, system, alert(level)) : norm_policy) ;
3    repeat (
4      unless done(S, system, alert(level)) :
5      if (done(S, system, alert(level)) then (
6              if (level = norm_level) then norm_policy
7         else if (level = high_level) then alert_policy
8      )
9    )
10 end
```

Initially the policy `norm_policy` is enforced, until the action `alert` is executed for the first time. This defines the initial default policy. After the security level has been set, the choice of policy is dependent on the currently set security level. This part of the policy is iterated, to capture that every execution of the action `alert` leads to a policy change[4]. Each iteration of the policy lasts until the next execution of the action `alert`. The enforced policy is conditional and depends on the actual security level, that was set.

This completes the operators that can be used for the temporal composition of policies. Another form of composition is the structural composition, where different policies can be specified to apply simultaneously.

7.4.2 Structural Composition

Often policies are used to control large-scale systems that span over several organisations. Here composition along the structural axis is useful to be able to capture the requirements for each unit (e.g. organisation, department, project-group, etc.) as an individual policy. The policy for a larger unit is then composed out of the policies defined for the smaller units, eventually yielding the system-wide policy. The main problem with this form of composition are conflicts between policies that can easily arise if the same resource or individual is placed under the control of more than one policy. Due to dynamics of the policies in the model the resolution of conflicts is sophisticated.

The basis for structural composition is the ability to define that a specific policy only applies to a subset of subject, objects and actions. These sets are referred to as the scope of the policy.

[4]The policy does even change, if the value of the level is not changed by the execution, viz. setting the security level from `norm_level` to `norm_level` represents a policy change. This affects the evaluation of the rules in the policies `norm_policy` and `alert_policy`, as the policy is *restarted*. If this is not the desired behaviour, the expression in the **unless** construct must be modified accordingly.

Scoping Policies

A specific scope can be specified at the definition of the policy itself. If the scope definition is omitted, it defaults to the universal scope, viz. the policy applies to all subjects, objects and actions in the system. The scope definition determines the range of the free variables S_i, O_i and A_i that can be used in rule definitions to reference subjects, objects and actions, respectively.

```
1 policy p : /* ... */ end
2 policy q : scope (any, [lpt1], any) : p  end
```

Here the policy `p` is defined without any restriction to the scope. The policy `q` is defined in terms of policy `p`, however, it limits its application to the object with the identifier `lpt1`. The keyword **any** denotes the respective universal set, i.e. S, O or A, dependent on the context. The multiple application of the scope operator does always constrain the scope of the policy. For example writing:

```
scope ([s1,s2],[o1,o2],[a1,a2]) : scope (any, [o1], [a2]) : p
```

is equivalent to writing:

```
scope ([s1,s2], [o1], [a2]) : p
```

Widening of the scope is *only* possible using the parallel composition of policies.

We assume that all policies that are composed sequentially have the *same* scope. If this is not the case, the scope of the individual policies must be further constraint or widened to meet this restriction.

Parallel Composition

Two policies can apply simultaneously. The typical case is that two policies define the requirements for separate parts of the system. This can lead to conflicts, if the scopes of both policies overlap. The following example illustrates this problem.

```
1  policy p :
2    ( allow (S,O,A) when true
3      decide (S,O,A) when allow(S,O,A) and not deny(S,O,A) )
4  end
5  policy q : scope ([s1], any, any)
6    ( allow (S,O,A) when true
7      deny (S,O,A) when A = write
8      decide (S,O,A) when allow(S,O,A) and not deny(S,O,A) )
9  end
10 policy pq : p and q end
```

The policy `p` states that any access is allowed. The second policy (`q`) states that subject `s1` is allowed any access, except `write`. The scopes of both policies overlap, as all accesses in the scope (`[s1],any,[write]`) are defined by both policies. The composition of both policies leads to an obvious conflict for the action `write` executed by the subject `s1`. In SANTA the parallel composition takes the strictest possible interpretation, viz. both policies must *explicitly* agree on a policy decision, for the decision to be made in their parallel composition.

Explicit agreement means that the decision must be derived by both of the component policies. In the example the policies `p` and `q` cannot agree on any decision for subjects other than `s1`, because they are not in the scope of *q*. This means that the policy `pq` does only authorise the subject `s1` to perform actions different to the action `write` on any object. All other subjects are denied by default, as the policy `q` does not specify any decision for them.

The view taken here is contrary to other policy models, where an *implicit* agreement is sufficient for the decision to be made in the composition. This means that the decisions for all subjects, other than `s1` are made as defined in policy `p`. For the above example this would mean that all subjects are allowed any access, with the exception of the subject `s1`, which cannot `write`. The reason for this different view is that implicit decisions are a danger during the specification process. This is comparable to a *"silent approval"*, viz. no statement means agreement with the decision of others. In SANTA not stating a decision means by default denying it. However, SANTA's parallel composition does allow to *explicitly* define the silent approval in form of an additional conflict resolution rule.

Given the strict interpretation of the parallel composition, a dedicated *simple policy* can be used resolve occurring conflicts and weaken the interpretation of the parallel composition. The conflict resolution policy may reference the scopes of both the policies, as well as their individual policy decisions. The notation for this is summarised below:

.subjects, .objects, .actions	lists of subjects, objects and actions that are in the scope of the left-hand side policy
subjects., objects., actions.	lists of subjects, objects and actions that are in the scope of the right-hand side policy
.allow, .deny, .decide, ...	reference the consequences of the left-hand side policy.
allow., deny., decide., ...	reference the consequences of the right-hand side policy.

The more lenient interpretation of policy composition can then be expressed as follows:

```
1  def inleftscope(x,y,z) :
2      (x in .subjects) and (y in .objects) and (z in .actions)
3  end
4
5  def inrightscope(x,y,z) :
6      (x in subjects.) and (y in objects.) and (z in actions.)
7  end
8
9  policy join :
10   ( decide (S,O,A) when 0: (inleftscope(S, O, A) and
11       not inrightscope(S, O, A) and .decide(S,O,A))
12
13     decide (S,O,A) when 0: (inrightscope(S, O, A) and
14       not inleftscope(S, O, A) and decide.(S,O,A))
15   )
16 end
17
18 policy pq : p and q deconflict join end
```

The simple policy `join` is used to weaken the parallel composition. The conflict resolution rules state that all those decisions, for subject, object and action pairings that are only defined in the scope of one of the component policies are fully determined by this policy. The first rule addresses the access for those triplets in the scope of `p` that are not in the scope of `q` (viz. (**any** \ [**s1**], **any**, **any** \ [**write**])). The second rule is the same, however, `p` and `q` are reversed.

The scope of the parallel composition of two policies is the union of the scopes of both policies. This allows to use the parallel composition of policies to explicitly widen the scope of a policy. This is a precondition to compose policies along the temporal axis.

```
1  policy p : scope ([s1],[o1],[a1]) /* */ end
2
3  policy leftpref :
4     ( decide (S,O,A) when inscope([S,O,A], .scope) and
5               .decide(S,O,A) )
6  end
7
8  policy q : p and () deconflict leftpref end
```

The policy `q` grants the same access as policy `p` within the scope (**any**,**any**,**any**). The conflict resolution policy `leftpref` defines that any decision that is made in the left-hand side policy must also be made in the composition. The empty policy () has the default scope of (**any**,**any**,**any**) and does not grant any access.

7.5 Semantics of Policy Composition

7.5.1 Temporal Composition

The semantics of the temporal composition has been defined by Siewe and is only included here for the completeness of the policy semantics. For a detailed discussion and proof rules see [124].

SANTA	Siewe's Notation	ITL
⟦P ; Q⟧	⟦P ; Q⟧	⟦P⟧ ; ⟦Q⟧
⟦aslongas expr : P⟧	⟦[expr]P⟧	((⟦P⟧ ∧ □expr) ; skip) ∧ fin ¬expr) ∨ (empty ∧ ¬ expr)
⟦unless expr : P⟧	⟦⟨expr⟩P⟧	⟦[¬ expr]P⟧
⟦expr : P⟧	⟦expr : P⟧	len = expr ∧ ⟦P⟧
⟦if expr then P else Q⟧	⟦expr?P : Q⟧	(expr ∧ ⟦P⟧) ∨ (¬ expr ∧ ⟦Q⟧)
⟦repeat P⟧	⟦(P)$^{\oplus}$⟧	⟦P⟧ ; (⟦P⟧)*

Table 7.2: Operators for the temporal composition of policies.

7.5.2 Structural Composition

Two operators that allow for the structural composition of dynamically changing policies have been introduced in the SANTA policy model: a) the scope operator that restricts the application of a rule to a specific subset of subjects, objects and actions and b) the parallel composition of two dynamically changing policies.

Policy Scope

Every policy has a scope that determines the set of subjects, objects and actions to which the policy applies to. By default the scope of a policy is the *universal scope*, viz. the set of all subjects S, objects O and actions A that can be controlled by policies. This means that the policy:

```
1 policy P :
2   ( decide(S,O,A) when true )
3 end
```

allows by default all subjects to perform all actions on all objects in the system. However, it is possible to constrain the application of this policy to a specific *scope*. For example:

```
1 policy P :
2   ( decide(S,O,A) when true )
3 end
4 policy Q :
5   scope (subjects, objects, [read]) : P
6 end
```

Here the policy Q is defined to behave exactly like policy P, however it is only applied to the action read. In this case the definition of P is trivial, and policy Q could be as easily defined from scratch, but the scope does also apply if P is a complex policy composition. The scope definition can be seen as an instantiation of the policy P for a specific set of subjects, objects and actions.

Explicit Scope Definition The scope operator allows for an explicit definition of policy scope. The explicit definition of a scope does affect the policy as it constraints the sets of subjects, objects and actions the policy is applied to. Any scope definition does reduce the set of subjects, objects and actions, by redefining these sets. The semantics of the scope operator is:

$$[\![\textbf{scope}\ (S_1, O_1, A_1)\ :\ \textsf{P}]\!] \widehat{=} [\![\textsf{P}]\!][(Subj \cap S_1), (Obj \cap O_1), (Act \cap A_1)/Subj, Obj, Act] \quad (7.27)$$

Every occurrence of the free set variables $Subj$, Obj and Act in P are substituted with the intersection of the set denoted by the free variable and the explicit scope (e.g. $Subj \cap S_1$). This does not bind the free variable, as it occurs freely in the substitution.

Referencing the Scope in Policies The set of subjects, objects and actions that are in the scope of a policy can be explicitly referenced in policies as the lists **subjects**, **objects** and **actions** or implicitly using the identifiers **S**, **O** and **A**.

Useful Theorems for the Scope The following theorems are useful for the manipulation of scopes.

Theorem 1
Scopes can be combined.

$$\vdash \textbf{scope}\ (S_1, O_1, A_1)\ :\ (\ \textbf{scope}\ (S_2, O_2, A_2)\ :\ \textsf{P}) \equiv \textbf{scope}\ (S_1 \cap S_2, O_1 \cap O_2, A_1 \cap A_2)\ :\ \textsf{P}$$

Proof 1
Proof outline: The individual scope definitions are expanded to their semantics. The substitution for the outer policy is performed. The result is transformed to show that it

is equivalent to the semantics of the scope with the intersected sets.

$$\textbf{scope}\ (S_1, O_1, A_1)\ :\ (\ \textbf{scope}\ (S_2, O_2, A_2)\ :\ \mathtt{P})$$

Substituting the inner scoped policy with the definition of the scope operator yields:

$$= \textbf{scope}\ (S_1, O_1, A_1)\ :([\![\mathtt{P}]\!][(S \cap S_2), (O \cap O_2), (A \cap A_2)/S, O, A])$$

Given the semantics of the inner scoped policy, the outer scope is expanded, too:

$$= ([\![\mathtt{P}]\!][(S \cap S_2), (O \cap O_2), (A \cap A_2)/S, O, A])[(S \cap S_1), (O \cap O_1), (A \cap A_1)/S, O, A]$$

Performing the substitution as defined by the outer policy yields:

$$= [\![\mathtt{P}]\!][((S \cap S_1) \cap S_2), ((O \cap O_1) \cap O_2), ((A \cap A_1) \cap A_2)/S, O, A]$$

The associativity of the set intersection is used to rearrange the order of the intersection:

$$= [\![\mathtt{P}]\!][(S \cap (S_1 \cap S_2)), (O \cap (O_1 \cap O_2)), (A \cap (A_1 \cap A_2))/S, O, A]$$

Given the definition of the scope operator this is the same as:

$$\textbf{scope}\ (S_1 \cap S_2, O_1 \cap O_2, A_1 \cap A_2)\ :\ \mathtt{P}$$

Theorem 2

A wider scope is absorbed.

$$S_1 \supseteq S_2 \wedge O_1 \supseteq O_2 \wedge A_1 \supseteq A_2 \vdash$$
$$\textbf{scope}\ (S_1, O_1, A_1)\ :\ \textbf{scope}\ (S_2, O_2, A_2)\ :\ \mathtt{P} \equiv \textbf{scope}\ (S_2, O_2, A_2)\ :\ \mathtt{P}$$

Proof 2

The proof of Theorem 2 follows the same lines as Proof 1. However, the subset relations $S_1 \supseteq S_2$, $O_1 \supseteq O_2$ *and* $A_1 \supseteq A_2$ *and the fact that* $X \supseteq Y$ *implies that* $X \cap Y = Y$ *(set theory) is used to remove the wider scope.*

Theorem 3

Scoping is commutative.

$$\vdash \textsf{scope}\ (S_1, O_1, A_1)\ :\ (\textsf{scope}\ (S_2, O_2, A_2)\ :\ \texttt{P}) \equiv \textsf{scope}\ (S_2, O_2, A_2)\ :\ (\textsf{scope}\ (S_1, O_1, A_1)\ :\ \texttt{P})$$

Proof 3

The proof of Theorem 3 follows the same lines as Proof 1. However, instead of the associativity of the set intersection, the commutativity is used.

Theorem 4

Scoping distributes over sequential composition and repetition.

$$\vdash \textsf{scope}\ (S_1, O_1, A_1)\ :\ (\texttt{P}\ ;\ \texttt{Q}) \equiv (\textsf{scope}\ (S_1, O_1, A_1)\ :\ \texttt{P})\ ;\ (\textsf{scope}\ (S_1, O_1, A_1)\ :\ \texttt{Q}))$$

$$\vdash \textsf{scope}\ (S_1, O_1, A_1)\ :\ (\textsf{repeat}\ \texttt{P}) \equiv \textsf{repeat}\ (\textsf{scope}\ (S_1, O_1, A_1)\ :\ \texttt{P})$$

Proof 4

Substituting S,O and A in the sequential composition $P\ ;\ Q$ is obviously the same as substituting in P and Q separately and sequentially composing them.

Theorem 5

Scoping distributes over **unless**, **aslongas**, *explicit time limit and the conditional if the expression does not reference the scope.*

$$\vdash \textsf{scope}\ (S_1, O_1, A_1)\ :\ (\textsf{unless}\ \textsf{expr}\ :\ \texttt{P}) \equiv (\textsf{unless}\ \textsf{expr}\ :\ (\ \textsf{scope}\ (S_1, O_1, A_1)\ :\ \texttt{P}))$$

$$\vdash \textsf{scope}\ (S_1, O_1, A_1)\ :\ (\textsf{aslongas}\ \textsf{expr}\ :\ \texttt{P}) \equiv (\textsf{aslongas}\ \textsf{expr}\ :\ (\ \textsf{scope}\ (S_1, O_1, A_1)\ :\ \texttt{P}))$$

$$\vdash \textsf{scope}\ (S_1, O_1, A_1)\ :\ (\textsf{expr}\ :\ \texttt{P}) \equiv (\textsf{expr}\ :\ (\ \textsf{scope}\ (S_1, O_1, A_1)\ :\ \texttt{P}))$$

$$\vdash \textsf{scope}\ (S_1, O_1, A_1)\ :\ (\textsf{if}\ \textsf{expr}\ \textsf{then}\ \texttt{P}\ \textsf{else}\ \texttt{Q}) \equiv$$

$$(\textsf{if}\ \textsf{expr}\ \textsf{then}\ \textsf{scope}\ (S_1, O_1, A_1)\ :\ \texttt{P}\ \textsf{else}\ \textsf{scope}\ (S_1, O_1, A_1)\ :\ \texttt{Q})$$

Proof 5

The proofs for Theorem 5 are all of a similar nature. The assumption is that the expression does not reference the policy scope. The fact that $\exists x \bullet (x = 0 \wedge w \wedge f) \equiv w \wedge \exists x \bullet (x = 0 \wedge f)$ given that x is not referenced in w is the foundation of all the proofs. The detail of the proofs is omitted.

Parallel Composition

The parallel composition of two policies means that both policies are enforced at the same time. This immediately leads to concerns about conflicts, that may occur if one policy defines a different decision than the other one.

Fundamental to Siewe's policy model is that it *guarantees* that policies are conflict free. The notion of a conflict is made explicit in the model by being able to define what a conflict means (for example if there is a positive and a negative authorisation for the same access) and resolving it using decision rules. Formally the fundamental reason why a policy cannot have conflicts in the traditional sense is that policy rules are defined as an implication and that no negation of the consequence is permitted.

This makes it more difficult to capture the intuition behind the parallel composition of policies. The following example illustrates:

```
1 policy P :
2 ( allow (S, money, take) when 0 : friend(S)
3   decide(S,O,A) when 0 : allow(S,O,A))
4 end
```

Policy `P` is a simple closed policy, where every access is denied unless explicitly allowed. The policy allows friends to take money. However, there may be another policy `Q`:

```
1 policy Q :
2 ( deny (S, money, take) when 0 : thief(S)
3   decide(S,O,A) when 0 : not deny(S,O,A))
4 end
```

This is an simple open policy, where every access is by default allowed, unless explicitly denied. In this case, the policy denies thieves to take money.

Assume that the parallel composition of these two policies would be defined using a simple conjunction. The obvious conflict in this example is when a friend is also a thief. In this case the decision rule of policy `P` would grant the access and policy `Q` would silently agree to this decision (because the decision rule is a form of implication). Although it is possible that the policy administrator has the intention to let friends take money irrespective of whether they are thieves, the point is that the model *cannot* decide differently. Siewe's addresses this issue for simple policies by filtering the rules and using set theoretic operations to remove the decision rules from the policies. However, for temporal compositions of policies the problem persists.

This example shows, that the simple conjunction of two policies does not capture the intuition behind parallel composition. SANTA defines a stricter parallel composition, that allows the policy administrator to define explicitly the resolution of conflicts. The intuition is that both policies are evaluated locally and the decision made by their parallel composition is derived from their individual decisions. The scope of the component policies is important for this as it defines the set of subjects, objects and actions each component policy was intended to apply to.

The parallel composition of policies is only defined for explicitly scoped policies. Using the Theorems 1 to 5, an explicit scope can be introduced for any policy. In the following definition *Conseq* denotes the control variables for the different types of rules (*autho*$^+$, *autho*$^-$, *autho*, *oblig* and *integ*).

$$
\begin{aligned}
&[\![\textsf{scope } (S_p, O_p, A_p) \textsf{ : P and scope } (S_q, O_q, A_q) \textsf{ : Q deconflict D}]\!] \mathrel{\widehat{=}} \\
&\forall s_p \in S_p, o_p \in O_p, a_p \in A_p, s_q \in S_q, o_q \in O_q, a_q \in A_q \bullet \\
&\exists Conseq_p(s_p, o_p, a_p), Conseq_q(s_q, o_q, a_q) \bullet \\
&(\quad [\![\textsf{scope } (S_p, O_p, A_p) \textsf{ : P}[Conseq_p(s_p, o_p, a_p)/Conseq(s_p, o_p, a_p)]]\!] \wedge \\
&\quad [\![\textsf{scope } (S_q, O_q, A_q) \textsf{ : Q}[Conseq_q(s_q, o_q, a_q)/Conseq(s_q, o_q, a_q)]]\!] \wedge \\
&\quad (\forall s \in Subj \bullet \forall o \in Obj \bullet \forall a \in Act \bullet (s \in S_p \cap S_q \wedge o \in O_p \cap O_q \wedge a \in A_p \cap A_q \wedge \\
&\qquad \mathsf{len} = 0 \wedge Conseq_p(s, o, a) \wedge Conseq_q(s, o, a) \mapsto Conseq(s, o, a))) \wedge \\
&\quad [\![D]\!])
\end{aligned}
$$

The parallel composition of the scoped policy P and a scoped policy Q introduces local boolean state variables $Conseq_p(s_p, o_p, a_p)$ and $Conseq_q(s_q, o_q, a_q)$ to capture the local policy decision of policy P and policy Q. s_p, o_p, and a_p range over P's scope, s_q, o_q, and a_q over Q's. The original policy definitions of P and Q are rewritten, to update these local variables. Since both policies are rewritten, to affect only local variables, their semantics can be safely conjoined. The definition states that for all subjects, objects, and actions that are in the scope of the parallel composition and that are in the scope of both policies a policy decision can be derived if the same decision can be derived from *both* component policies.

This means that both policies must explicitly agree on a specific policy decision, for the decision to be made in their composition. Any decision that is taken unilaterally by one policy will not be reflected in the composition. This strict interpretation of parallel composition, does not capture the intuition of the parallel composition, either. For example the parallel composition of two policies with completely disjoint scopes leads to a policy, that does not grant any access, defines no obligations and imposes no integrity constraints. However, using the conflict resolution policy D the interpretation of the parallel composition can be relaxed.

In the conflict resolution policy D the original scopes of the policies P and Q, as well as their local policy decisions can be referenced. In the syntax, all references to **subjects**, **objects** and **actions**, as well as to the policy decisions **allow**, **deny**, **decide**, **oblige** and **integrity** that are prefixed with a dot reference the local scope and decisions of the left-hand side

policy of the composition. All those that are suffixed with a dot the local scope and decisions of the right-hand side policy. The keywords to reference the decisions of the parallel composition and the respective local decisions and the variables they represent in the definition of the parallel composition are summarised in Table 7.3.

SANTA			Corresponding Variable		
subjects	.subjects	subjects.	$Subj$	S_p	S_q
objects	.objects	objects.	Obj	O_p	O_q
actions	.actions	actions.	Act	A_p	A_q
allow(x,y,z)	.allow(x,y,z)	allow.(x,y,z)	$autho^+(x,y,z)$	$autho_p^+(x,y,z)$	$autho_q^+(x,y,z)$
deny(x,y,z)	.deny(x,y,z)	deny.(x,y,z)	$autho^-(x,y,z)$	$autho_p^-(x,y,z)$	$autho_q^-(x,y,z)$
decide(x,y,z)	.decide(x,y,z)	decide.(x,y,z)	$autho(x,y,z)$	$autho_p(x,y,z)$	$autho_q(x,y,z)$
integrity(x,y,z)	.integrity(x,y,z)	integrity.(x,y,z)	$integ(x,y,z)$	$integ_p(x,y,z)$	$integ_q(x,y,z)$
oblige(x,y,z)	.oblige(x,y,z)	oblige.(x,y,z)	$oblig(x,y,z)$	$oblig_p(x,y,z)$	$oblig_q(x,y,z)$

Table 7.3: Referencing local decisions

The use of the dot as a prefix or suffix is only permitted in the context of simple policies that de-conflict the parallel composition. Using the de-confliction policy the strict definition of the parallel composition can be relaxed. Examples of this have been provided earlier on in Section 7.4.2.

7.6 Summary

This chapter introduced the syntax and semantics of SANTA policies. Two conceptional different types of policies can be expressed: a) *environmental* policies that constrain the access to interfaces of SANTA objects and b) *behavioural* policies that constrain the behaviour of SANTA agents. Both types of policies can be distinguished by the subjects, objects and actions they apply to. For an environmental policy the controlled objects are SANTA objects and the controlled actions the interfaces that they expose. For behavioural policies the objects are SANTA agents and the controlled actions the actions that are encapsulated within the agents.

SANTA policies express security requirements, such as authorisation, delegation, integrity and obligation, in form of rules. Each rule defines the type of requirement and the system entities that are affected by it in form of a consequence. The premise of a rule defines a set of behaviours that lead to the consequence. The possibility to define a behaviour in the premise allows for the expression of complex history dependent requirements, such as the Chinese Wall Policy. Examples of the various type of rules as well as state and history based requirements have been provided in Sections 7.2.1 to 7.2.5.

Policy rules are combined into *simple policies*, representing a set of policy rules that

apply simultaneously. One of the key advantages of the policy model is that policies can be composed to capture the dynamic change of policies based on time and the observation of events. This enables the policy administrator to focus during the specification of a policy on specific situations and then define the events and conditions that describe the transition between them. This is captured by the *temporal composition* of policies. An example of a policy that depends on a system wide threat level has been given in Section 7.4.1. The *structural composition* of policies allows to restrict a policy to a specific scope and to combine temporal compositions to apply simultaneously. The structural composition is used to combine policies that are defined for different units within an organisational hierarchy to obtain the policy that applies to the overall organisation.

The semantics of policies has been defined in Section 7.3 and 7.5. It was shown, how interface parameters can be externalised and be made accessible in the specification of policy rules. The use of prophecy variables to reference the values of parameters at a specific state in the history has been detailed. The semantics of rules has been gradually developed to take into account state and parameter dependencies.

At the semantic level, policies define the behaviour of policy specific control variables over a sequence of states. These control variables are modelled as boolean ITL state variables that capture the policy decision in every state.

Chapter 8

Enforcement

In this chapter we introduce the mechanisms that are available in SANTA for the enforcement of policies. The enforcement mechanisms link the the SMAS with policies.

8.1 Introduction

Enforcement mechanisms define how policies are enforced in the system. Because a SMAS is a highly distributed system we cannot assume the existence of one centralised mechanism that is enforcing the policy. Instead we define three different mechanisms that can be distributed in the system. These are *vigilant agent*, *vigilant object* and *security enforcer*.

Distributed Enforcement Mechanisms For a vigilant agent the enforcement is part of the agent's behaviour. It is implemented in the *enforcement phase* of the agent and as part of the encapsulated actions. A vigilant agent can enforce behavioural policies for the agent. Similarly, for a vigilant object the enforcement is part of the object's behaviour. It is implemented as part of the interfaces that the object provides. A vigilant object can enforce environmental policies for the object. The security enforcer is similar to the vigilant object mechanism, however it controls the access and integrity of a group of objects. The enforcement mechanism mediates the access to the interfaces of the objects under protection. The security enforcer can enforce environmental policies for *a group* of objects.

The link between policies and system The semantics of objects and agents defines their behaviour as a set of sequences of states. In SANTA this specification is initially defined at a higher level of abstraction and then refined to be concrete and implementable. This is depicted in Figure 8.1 by the lower two state sequences. At the higher abstraction level, the definition of actions and interfaces state only the effect of the computation, not how the computation is actually performed or how many states are required. During the development the abstract specification is refined into a concrete specification that introduces additional states (depicted by the lower state sequence).

Similarly a policy defines the behaviour of the enforcement variables over a sequence of states, viz. the access control decisions, obligations and integrity in each state. The policy specification is more abstract than the specification of the agents and objects as it is only defined in terms of subjects, objects and actions. The implementation details of the actions or interfaces are hidden and not of interest at this level of abstraction. The policy specification has an even coarser grained view of the system. This is depicted in Figure 8.1.

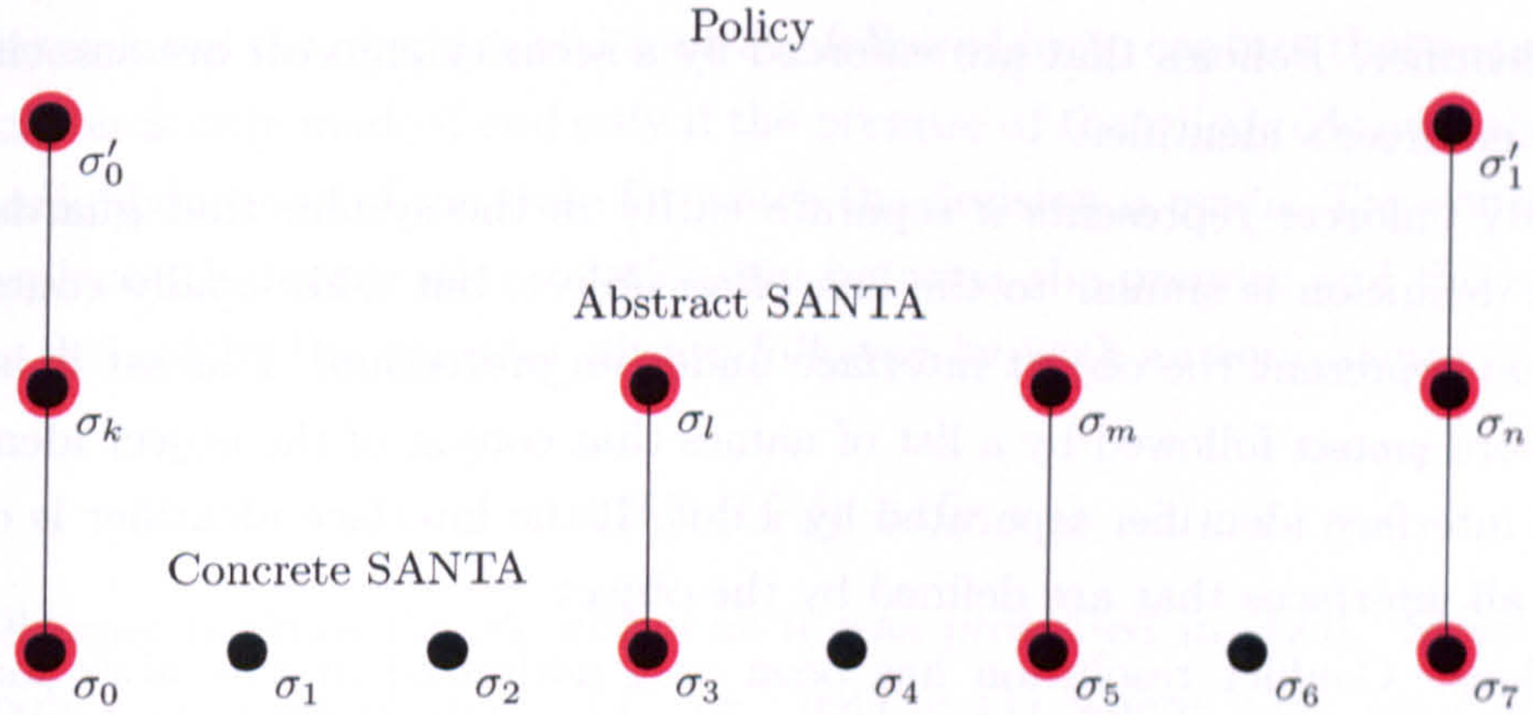


Figure 8.1: Abstraction Levels

The semantics of the enforcement mechanisms defines the mapping between the states over which the policy is defined and those in the abstract specification of agents and objects as defined by the $SMAS_2$. Additionally they relate the control variables in the $SMAS_2$ to the enforcement variables in *enforcement properties*. The enforcement properties define what the enforcement of a policy means at a high abstraction level, viz. over the same sequences of states over which the policies are defined.

In this chapter we are concerned with the abstract specification of the enforcement mechanisms, viz. what does it mean for a policy to be enforced and how do policies influence the behaviour of agents and objects. The development of mechanisms that ensure that the overall system does not violate the properties stated by the enforcement mechanisms is discussed subsequently in Chapter 9.

In Section 8.2 we introduce the syntax that is used to define enforcement mechanisms, and to associate policies with them. In Section 8.4 we show how a policy can be transformed into a *complete* (deterministic) policy. Section 8.5 and 8.6 define the meaning the vigilant mechanisms; Section 8.7 the semantics of the security enforcer.

8.2 Enforcement Mechanisms in SANTA

```
1  enf  =  'enforce', policy, 'with', id .
2  senf =  'securityenforcer', id , ':',
3          ( 'protect' id, ['.', id] { ',', id, ['.', id] } ),
4          { vardecl }, { intdecl }, 'end' .
```

The keyword **enforce** defines the link between a policy and an enforcement mechanism. `id` is the identifier of an object, agent or security enforcer that enforces the policy. For vigilant agents the policy is associated with an agent identifier; for vigilant objects with

an object identifier. Policies that are enforced by a security enforcer are associated with the security enforcer's identifier.

A security enforcer represents a separate entity in the system that guards a set of objects. Its definition is similar to the one of an object, but additionally contains a set of names that represent the object interface under its protection. This set is introduced by the keyword **protect** followed by a list of names that consist of the object identifier and an optional interface identifier separated by a dot. If the interface identifier is omitted it expands to all interfaces that are defined by the object.

Limitations: Conflict resolution has been only addressed in this work for policies. However conflicts can also occur between enforcement mechanisms. This is not addressed in this work, but can be defined along the same lines as the conflict resolution of policies. We assume in the following that any object is protected by exactly one enforcement mechanism.

8.3 Subjects, Objects and Actions

In Chapter 7 we defined the set variables for subjects *Subj*, objects *Obj* and actions *Act* to be free in the policy specification. The enforcement mechanisms combine the semantics of policies with the semantics of the system and therefore need to bind these free variables to concrete sets that are meaningful w.r.t to the system definition. The binding of the free set variables is defined as follows:

$$bind \mathrel{\widehat{=}} (Subj = \mathcal{A}) \wedge (Obj = \mathcal{A} \cup \mathcal{O}) \wedge (Act = (\bigcup_{a \in \mathcal{A}} \bigcup_{x \in X_a} x) \cup (\bigcup_{o \in \mathcal{O}} \bigcup_{i \in I_o} i)) \tag{8.1}$$

The set of all subjects in the policy specification is the set of all SANTA agents in the SMAS. The objects in the policy specification can be either agents or objects in the SMAS. The actions that can be controlled by policies are either the agent actions (for behavioural policies) or they are interfaces (for environmental policies).

8.4 Complete Policy Specification

Any security policy that is enforced by any of the mechanisms that are presented in the following sections must fully specify the outcome of all policy decisions for the scope that is controlled by the enforcement mechanism. Siewe presented in [124] an algorithm that transforms a the specification of a policy to a complete policy. The algorithm guarantees that the policy specification determines the value of the enforcement variables in every state of the interval over which the policy is defined.

Siewe introduced the operator *strict always followed by* to capture that a specific access control decision is only made if and only if the premise of the rule holds over some interval in the left neighbourhood of the state for which the decision is made. The definition of the operator replaces the implication relationship between the premise and the consequence of a rule as defined by the operator *always followed by* with an equivalence:

$$f \leftrightarrow w \mathrel{\widehat{=}} \boxdot ((\Diamond f) \equiv (\mathsf{fin}\ w)) \tag{8.2}$$

The following outlines the algorithm as it was presented in [124]. Siewe represents a simple policy as a set of rules: $\{f \mapsto op(x, y, z)\}$ where x,y,z are constants and $op \in \{autho, autho^+, autho^-\}$. All rules in the set hold in conjunction. We expand the set op to include additionally *oblig* and *integ*. The set of rules can be easily produced from the semantics of a simple policy that was provided in Section 7.3.4. First the bounded universal quantification is expanded into an equivalent conjunction of the form:

$$f_0 \mapsto op(x_0, y_0, z_0) \wedge \ldots \wedge f_{l,m,n} \mapsto op(x_l, y_m, z_n)$$

x_i, y_j and z_k are constants identifying concrete subjects, objects and actions. The set of all rules, in the following referred to as p, is produced by splitting the conjunction of the individual always-followed-by operators.

The algorithm takes as input a simple policy p, the set of subjects $X \subseteq Subj$, the set of objects $Y \subseteq Obj$ and the set of actions $Z \subseteq Obj$ with respect to which the policy p is completed. The result is a policy p' for which all policy decisions are specified. The algorithm guarantees that any decision (assignment of the corresponding enforcement variable to true) that is made in p is also made in p'.

Step 1. Construct the set

$$p_1 = p \cup \bigcup_{x \in X, y \in Y, z \in Z} \{\mathsf{false} \mapsto op(x, y, z)\}$$

The semantics of $\mathsf{false} \mapsto op(x, y, z)$ is true, i.e. the conjunction of all rules in p_1 is equivalent to the conjunction of all rules in p.

Step 2. Regrouping of rules. Siewe provides a Theorem that states the conjunction of two rules with the same consequence is equivalent to a single rule with the same consequence, where the premise is the disjunction of the premises of the original rules:

$$(f_1 \mapsto w) \wedge (f_2 \mapsto w) \equiv ((f_1 \vee f_2) \mapsto w)$$

This is used to regroup the rules in the set to a normal form, where every consequence occurs *exactly once*.

$$p_2 = \bigcup_{x \in X, y \in Y, z \in Z} \{f^{op}_{x,y,z} \mapsto op(x,y,z)\}$$

where $f^{op}_{x,y,z} = f^{op}_1 \vee \ldots \vee f^{op}_n$ is the disjunction of all the f^{op}_i such that $f^{op}_i \mapsto op(x,y,z) \in p_1, i = 1,\ldots,n$.

Step 3. Siewe shows that the operator strict-always-followed-by is a refinement of the operator always-followed-by, viz. that $(f \leftrightarrow w) \supset (f \mapsto w)$. In this step the operator *always followed by* is replaced with the operator *strict always followed by*. This is denoted by the set p' that is the result of the algorithm.

$$p' = \{f^{op}_{x,y,z} \leftrightarrow op(x,y,z)\}$$

Given that all rules in the set p' apply in conjunction, this fully defines the policy decisions for all the control variables. The result of the process is the same as assuming that the default value for the case that no decision can be derived is false.

The algorithm $Comp(X,Y,Z,p)$ takes as input the set of subjects X, the set of objects Y, the set of actions Z and a simple policy p and produces as output the complete policy p'. This can be easily extended to composed policies as described in [124]. The enforcement mechanisms presented in the following will use the *Comp* algorithm to ensure the completeness of the enforced policy with respect to the subjects, objects and actions in their maximal enforceable scope.

8.5 Vigilant Agent

A *vigilant* agent implements enforcement mechanisms directly within the agent's *enforcement phase* and as part of the action execution. An agent a is defined to be vigilant by associating a policy P_a with the agent:

```
1  enforce P_a with a
```

8.5.1 Enforceable Policies

A vigilant agent can enforce behavioural policies for the agent itself. Formally this means that the scope of policies that can be enforced by the vigilant agent a is

$$(Subj_a, Obj_a, Act_a) = (\{a\}, \{a\}, X_a)$$

Associating a policy with a vigilant agent implicitly scopes the policy to this maximal enforceable scope. This means that the above association of the policy p with the agent a is equivalent to:

```
1  enforce scope ({a},{a},X_a) : P_a with a
```

Where X_a is the set of actions encapsulated in the agent a. Each vigilant agent can enforce exactly one policy. If more that one policy is to be enforced then the policies must be explicitly composed using the operators presented in Chapter 7. The type of rules that the policy can contain are *authorisation*, *obligation* and *integrity*. Delegation rules are not in p's scope, because the agent *cannot* define the interfaces `delegate` and `revoke`.

The policy may reference the agent variables encapsulated in the agent a by their identifier, or the fully qualified name *a.id*. The policy can also reference the control variables $done_{a,x}$ and $failed_{a,x}$.

8.5.2 Mapping Policy- and System States

A policy that is enforced by a vigilant agent is interpreted over an interval that contains only those states in which the vigilant agent can start the execution of an action. These are the states in the interval defined by $SMAS_2$ for which $ready_a$ is *true*.

We know that from one of these states to the next the agent executed at most one action. Consequently the agent variables have been assigned at most *once* as the computation takes place in local (or auxiliary) variables and the results are only copied back to the agent variables if the execution is successful. Events that are defined in terms of agent variables cannot be *lost* in this coarser view. This is depicted in Figure 8.2.

If the execution of action x by the vigilant agent a in state σ_0 did update the agent variables, then any event that is defined as an expression on these variables is still observable in the subsequent state where the agent is *ready* again (In Figure 8.2 this is the state σ_3). The mapping from these states to the policy states σ'_0 and σ'_1 ensures that from one policy decision to the next no policy related events are lost.

Defining the policy decisions only for the states in which the agent is ready is sufficient, as the agent can only start the execution of an action in this state. This shows

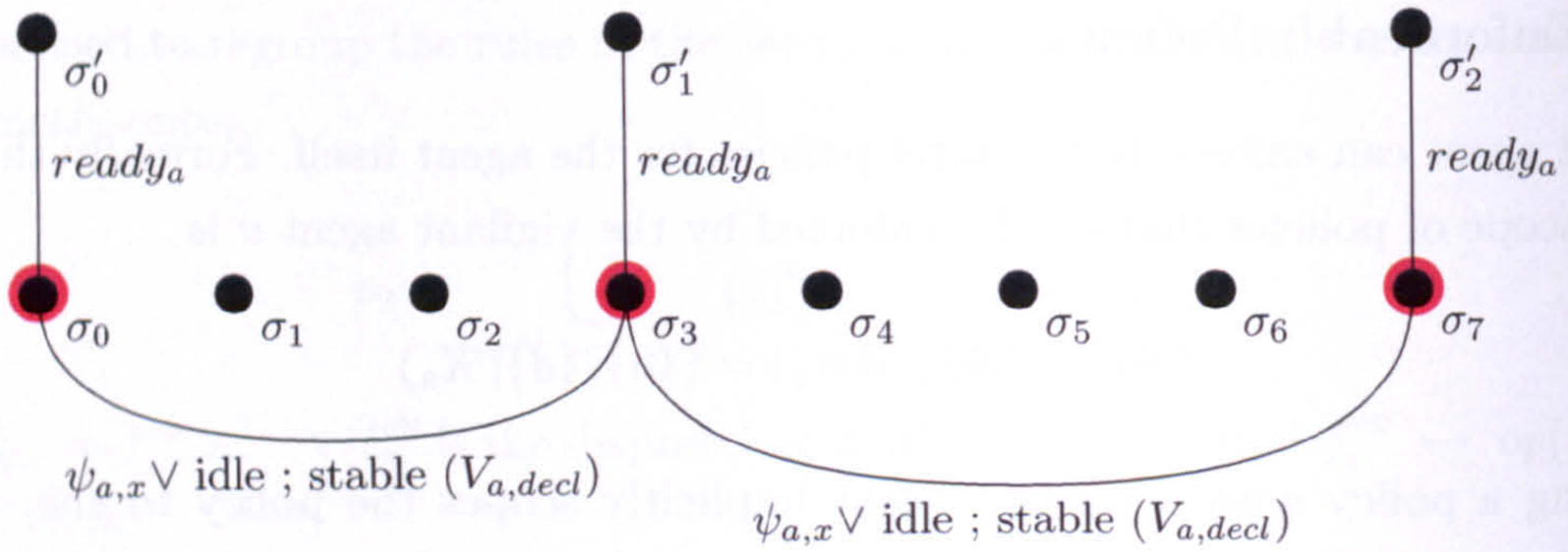


Figure 8.2: State projection for vigilant Agent

that selecting only the states where the agent is ready is a suitable abstraction for the enforcement of behavioural policies.

The interval σ', containing only the states for which $ready_a$ is *true*, is obtained using the ITL projection operator $S_a \Delta T_a$. T_a is the formula that holds on the projected interval and S_a the formula that describes the intervals bridging between the projected states. In the case of the vigilant agent enforcer a, the formula S_a bridges between two states in which the agent a is *ready*:

$$S_a = ((ready_a \wedge \mathsf{skip}) \,;\, ((\mathsf{keep} \neg ready_a) \wedge \mathsf{fin}\ ready_a)) \tag{8.3}$$

Equation (8.3) describes an interval that is at least of length one, in which $ready_a$ is *true only* in the initial and the final state of the interval.

The formula T_a that must hold over the projected interval is the complete policy P'_a with respect to the maximal enforceable scope of the vigilant agent a.

$$P'_a \mathrel{\widehat{=}} Comp(Subj_a, Obj_a, Act_a, P_a) \tag{8.4}$$

The complete policy P'_a holds in conjunction with the enforcement properties $EP_{a,autho}$, $EP_{a,oblig}$ and $EP_{a,integ}$ that define the enforcement of the respective policy class, viz. the effect that a policy decision has on the system. They are defined subsequently in subsections 8.5.3, 8.5.4, and 8.5.5.

$$T_a = P'_a \wedge EP_{a,autho} \wedge EP_{a,oblig} \wedge EP_{a,integ} \tag{8.5}$$

The semantics of a SMAS_3 that contains policies P_a that are enforced by the vigilant

agents $a \in SE_{vigagent}$ is then defined as a conservative extension to the SMAS_2 subset:

$$SMAS_3 \mathrel{\widehat{=}} SMAS_2 \wedge bind \wedge \bigwedge_{a \in SE_{vigagent}} (\mathsf{keep}\,(\neg\, ready_a))\,;(S_a \Delta T_a)) \tag{8.6}$$

$SE_{vigagent}$ defines the set of agents that enforce a policy vigilantly. $SE_{vigagent}$ is a subset of the set of all agents $\mathcal{A}$. The prefix interval $(\mathsf{keep}\,(\neg\, ready_a))$ bridges the start of the agent and the start of the policy enforcement, viz. the policy enforcement starts in the first state in which $ready_a$ is *true*. The link between the SMAS_2 and the enforcement is depicted in Figure 8.3.

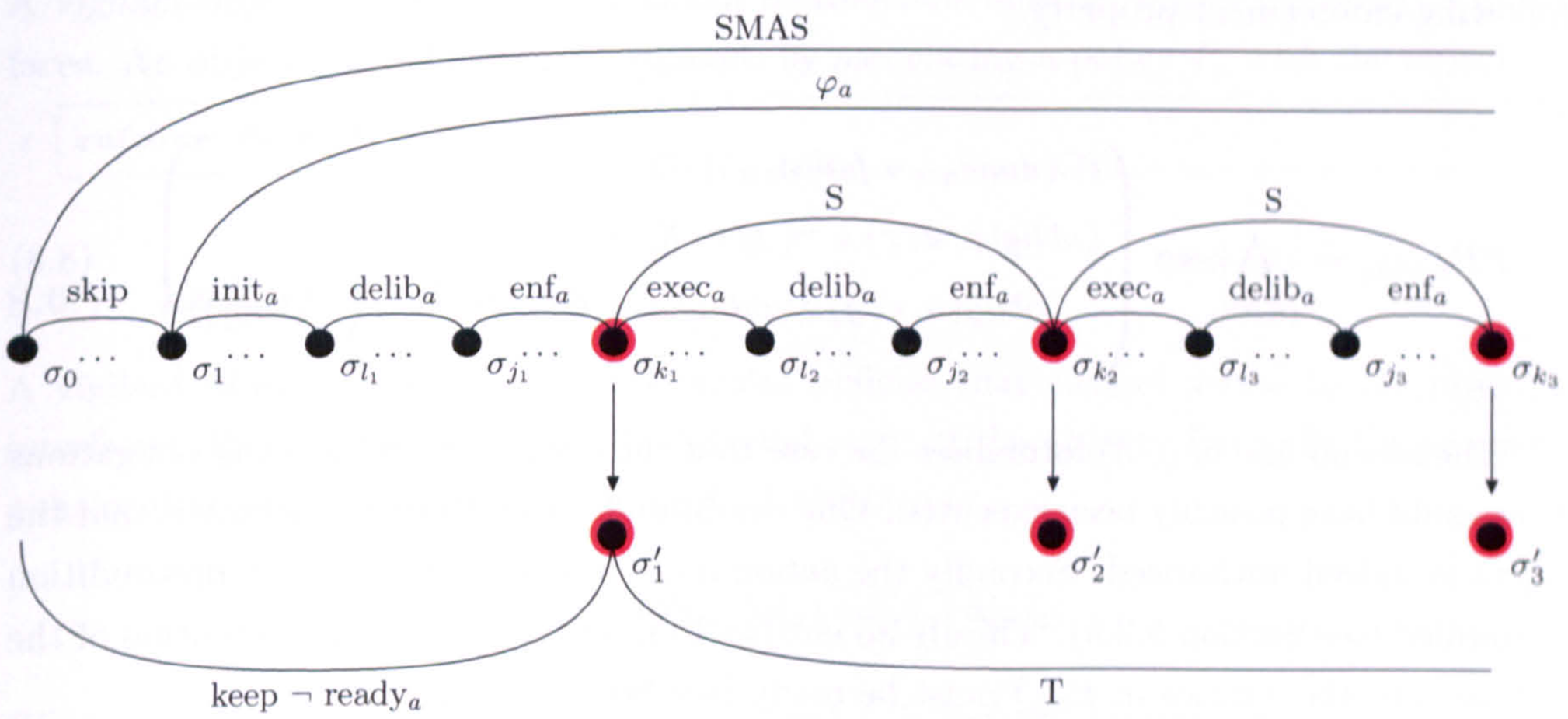


Figure 8.3: Vigilant Agent enforcing a policy

The formula $(\mathsf{keep}\,(\neg\, ready_a))$ bridges the time until the agent first enters the execution phase. The states in which the agent enters the execution phase are labelled σ_{k_i} (highlighted in red). In these states the control variable $ready_a$ is *true*. The interval between the states σ_{k_i} and $\sigma_{k_{i+1}}$ is bridged by the formula S_a, defining the projected interval $\sigma' = \sigma'_1, ..., \sigma'_n$ over which the policy and the enforcement properties are specified.

8.5.3 Enforcing Authorisation

The effect that the enforcement of authorisation policies by a vigilant agent has on the system execution is defined by the enforcement property $EP_{a,autho}$. Informally the enforcement of authorisation means that if an agent a is not permitted to perform an action x in a state σ'_i, then it cannot be the case that the agent a has successfully executed the action x in the next state σ'_{i+1}. This is expressed in (8.7).

$$EP_{a,autho} \mathrel{\widehat{=}} \bigwedge_{x \in Act_a} \mathsf{keep}\,(\neg\, autho(a,a,x) \supset \bigcirc \neg\, done_{a,x}) \tag{8.7}$$

8.5.4 Enforcing Obligations

The link between an obligation in the policy model and the SMAS system is defined similarly to the enforcement of authorisations. Informally the enforcement of an obligation means that if an agent a executed an action x in state σ'_{i+1}, then it is either the case that the agent has been obliged to execute this action in the previous state σ'_i, or the agent did not have any obligations that could have possibly been executed. This is captured by the following enforcement property:

$$EP_{a,oblig} \mathrel{\widehat{=}} \bigwedge_{x \in Act_a} \mathsf{keep} \left(\begin{array}{l} (\bigcirc(done_{a,x} \vee failed_{a,x})) \supset \\ (oblig(a,a,x) \vee \neg(\exists y \in X_a \bullet \\ \quad oblig(a,a,y) \wedge autho(a,a,y) \wedge p_{a,y} \wedge \bigwedge_{j \in E_{a,y}} ready_j)) \end{array} \right) \tag{8.8}$$

The second line in (8.8) formalises the case that the agent did not have any obligations that could have possibly been executed. One condition for the ability to execute is that the agent is indeed authorised. Secondly the action must be executable, viz. its precondition is fulfilled (see Section 5.3.5). Thirdly all entities that are involved in the execution of the action (viz. the entities in $E_{a,y}$) must be ready (see Section 6.3.1).

Other work [126, 117, 39] takes a much stronger view of obligation, where an obligation guarantees that a specific action is executed or a specific state of the environment is brought about. However, these notions represent a property of the system that in most cases is checked statically against the system specification. This is in principle also possible within the presented framework, but not addressed in this work.

The view of obligation taken here is less strict and has a more operational interpretation, viz. an obligation can overrule the decision that was made in the deliberation phase of an agent. This is actually enforceable by reassigning the priorities for the agent's actions and captures the intuition that is behind the enforcement of behavioural policies.

8.5.5 Enforcing Integrity

The enforcement of integrity policies is similar to authorisation policies. Informally the enforcement of integrity constraints means that no action can have successfully executed, if its integrity constraints have not been met. This is defined as:

$$EP_{a,integ} \mathrel{\widehat{=}} \bigwedge_{x \in Act_a} \Box(\neg\, integ(a, a, x) \supset \neg\, done_{a,x}) \tag{8.9}$$

If the integrity check has failed then the action cannot have succeeded. Note that here $integ(a, a, x)$ and $done_{a,x}$ are evaluated in the same state, because the results of the execution of x are only available at the end of the action execution. They are accessible through the auxiliary variables for that action (see Section 7.3.3).

8.6 Vigilant Object

A *vigilant object* implements enforcement mechanisms directly within the object's interfaces. An object o is defined to be vigilant, by associating a policy P_o with the object:

```
1 enforce P_o with o.
```

8.6.1 Enforceable Policies

A vigilant object can enforce environmental policies that control access to the object's interfaces. Policies may reference the internal state of the object. Formally the scope of policies that can be enforced by the vigilant object o is:

$$(Subj_o, Obj_o, Act_o) = (\mathcal{A}, \{o\}, I_o)$$

Where $\mathcal{A}$ is the set of agents in the systems and I_o is the set of interfaces that is provided by the vigilant object o. By associating a policy with a vigilant object, the scope of the policy is implicitly limited to the maximal enforceable scope, viz. the above association is equivalent to writing:

```
1 enforce scope (A,{o},I_o) : P_o with o.
```

The type of policy rules that can be enforced by this mechanism are authorisation, delegation and integrity policies. Obligation policies are not enforceable, because the object is a passive entity that cannot initiate any computation. Policies may reference o's object variables.

8.6.2 Mapping Policy- and System States

In Chapter 7 we assumed that the enforcement of environmental policies does guarantee that the parameter values that are passed during the interface invocation have been assigned to their corresponding auxiliary variable *before* the policy decision is made. This

means that we cannot use the states in which the objects are *ready* for the projection, because these are just *before* the auxiliary variables are set. To define the state in which all variables that are required for the interface execution are initialised, we introduce a marker variable enf_o.

Introducing enf_o as a marker

The initialisation of the local variables (respectively the auxiliary variables) is defined by $init_{i,x}$ in (6.7) on page 137. To be able to identify the state in which the initialisation has finished we introduce a boolean marker variable enf_o for each object o, that is only true in exactly one state after the variables have been initialised. Theorem 6 states that the introduction of a new variable that acts as a marker is a refinement (see Chapter 9).

Theorem 6

The introduction of a new boolean state variable that acts as a marker is a refinement. The marker variable can be used to mark states that are directly before or after a temporal assignment and is valid after the initialisation of the object. Provided the statement $[x \leftarrow y]_V$ *is part of the specification of an object in the SMAS and* $V \supseteq V_{o,contr}$ *then replacing the statement with*

$$[x \leftarrow y\ ;\ m{:=}\mathsf{true}\ ;\ m{:=}\ \mathsf{false}]_{V'}$$

or with

$$[m{:=}\mathsf{true}\ ;\ m{:=}\ \mathsf{false}\ ;\ x \leftarrow y]_{V'}$$

is a refinement. Where V' *denotes the set of augmented object state variables. To fully specify the behaviour of the marker variables the specification is strengthened to ensure that it is kept stable in all other states. To ensure this, the set of control variables is substituted in the object specification with the augmented set* $V'_{o,contr} = V_{o,contr} \cup \{m\}$ *after the initialisation phase of the object* $init_o = \overline{v} \leftarrow \overline{e}$ *is refined to* $init'_o = \overline{v}, m \leftarrow \overline{e}, \mathsf{false}$.

Proof 6

The proof for this theorem very similar to the Proof 8 in Chapter 9 and not repeated here. Since the marker is a freshly introduced variable, its assignment does not modify any variables in V*. By adding the marker to the set of variables, we complete the specification w.r.t. the marker.*

Applying the theorem we introduce the marker enf_o for every object o as a suffix of the interface initialisation $init_{i,x}$:

$$init_{i,x} \sqsubseteq [\![v'_0, \ldots, v'_k, p_0, \ldots, p_l \leftarrow v_0, \ldots, v_k, e_0, \ldots, e_l ;$$
$$enf_o := \mathsf{true}; enf_o := \mathsf{false}]\!]_{V^+ \setminus \{p'_0, \ldots, p'_l\} \cup \{enf_o\}}$$

The set of control variables for the object o is extended by the variable enf_o to ensure that the variable is kept stable by the rest of the object specification.

Mapping to the marker

The enforcement of policies is defined analogously to the vigilant agent case. However, the policy is now projected on the states in which enf_o is *true*.

$$S_o = (enf_o \wedge \mathsf{skip}) \,;\, ((\mathsf{keep}\ \neg\, enf_o) \wedge \mathsf{fin}\ enf_o) \tag{8.10}$$

S_o is the formula that bridges between the selected states. The complete policy P'_o with respect to the maximal enforceable scope of the mechanism is:

$$P'_o \mathrel{\widehat{=}} Comp(Subj_o, Obj_o, Act_o, P_o) \tag{8.11}$$

The policy P'_o that is enforced by the vigilant object o together with the enforcement properties for authorisation and integrity is defined as T_o.

$$T_o = P'_o \wedge E_{o,autho} \wedge E_{o,integ} \tag{8.12}$$

The semantics of a SMAS_3 that contains policies P_o that are enforced by the vigilant objects $o \in SE_{vigobject}$ is then defined as a conservative extension to the definition given in (8.6):

$$SMAS_3 \mathrel{\widehat{=}} SMAS_2 \wedge bind \wedge \tag{8.13}$$
$$\bigwedge_{a \in SE_{vigagent}} (\mathsf{keep}\,(\neg\, ready_a)) \,;\, (S_a \Delta T_a)) \wedge$$
$$\bigwedge_{o \in SE_{vigobject}} ((\mathsf{keep}\,(\neg\, ready_o)) \,;\, (\mathsf{keep}\,(\neg\, enf_o)) \,;\, (S_o \Delta T_o))$$

Here $SE_{vigobject}$ denotes the set of all vigilant objects that are defined in the SMAS. The formula $\mathsf{keep}\,(\neg\, ready_o)$ bridges the initialisation phase of the object, and the formula $\mathsf{keep}\,(\neg\, enf_o)$ over the subsequent interval until the variables for the first interface invocation on object o are initialised.

The level of abstraction that is provided by this mapping guarantees, that no policy related event, that is specified in terms of object variables, can be lost. Object variables are only modified by interfaces. The marker enf_o is *true* after the initialisation of each interface execution. Consequently from one state σ'_i in the abstraction to the next state σ'_{i+1} all policy related changes can be observed. The abstraction is also intuitive, as it means that one unit interval in the policy specification bridges the execution of exactly one interface at the system level. It also ensures that for every interface execution a policy decision is made.

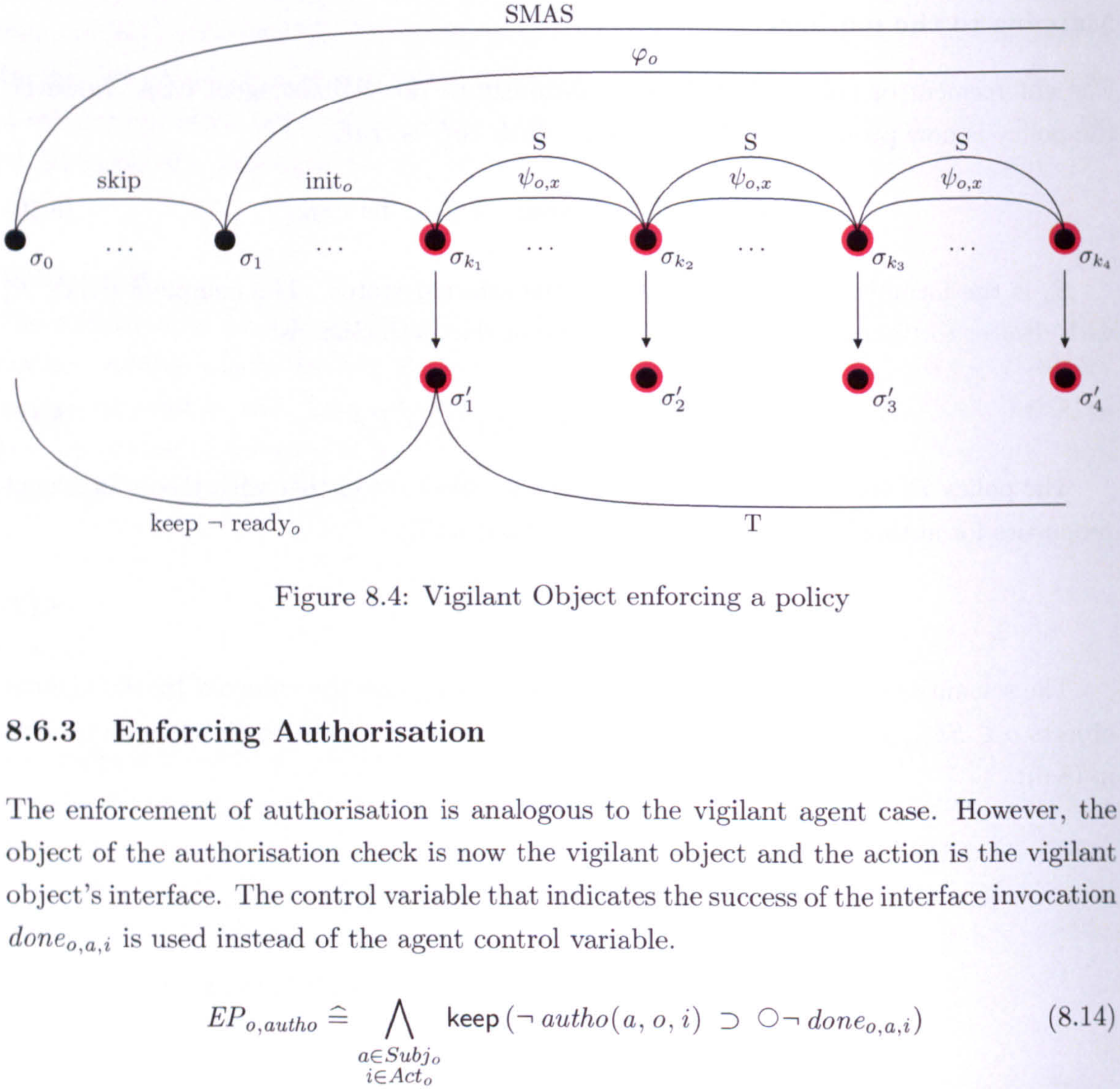


Figure 8.4: Vigilant Object enforcing a policy

8.6.3 Enforcing Authorisation

The enforcement of authorisation is analogous to the vigilant agent case. However, the object of the authorisation check is now the vigilant object and the action is the vigilant object's interface. The control variable that indicates the success of the interface invocation $done_{o,a,i}$ is used instead of the agent control variable.

$$EP_{o,autho} \mathrel{\widehat{=}} \bigwedge_{\substack{a \in Subj_o \\ i \in Act_o}} \mathsf{keep}\,(\neg\, autho(a,o,i) \supset \bigcirc \neg\, done_{o,a,i}) \tag{8.14}$$

The enforcement property states that if the agent a is not authorised to invoke the interface i of object o in state σ'_i then it cannot be the case that in the next state σ'_{i+1} the interface has been successfully executed by a.

8.6.4 Enforcing Integrity

The enforcement of integrity is analogous to the vigilant agent case. It states that if the integrity constraints are not met, then the interface invocation cannot be successful.

$$EP_{o,integ} \mathrel{\widehat{=}} \bigwedge_{\substack{a \in Subj_o \\ i \in Act_o}} \Box(\neg\, integ(a, o, i) \supset \neg\, done_{o,a,i}) \tag{8.15}$$

For integrity policies the level of abstraction poses a special update problem if the policies reference parameters. This is illustrated in Figure 8.5 below.

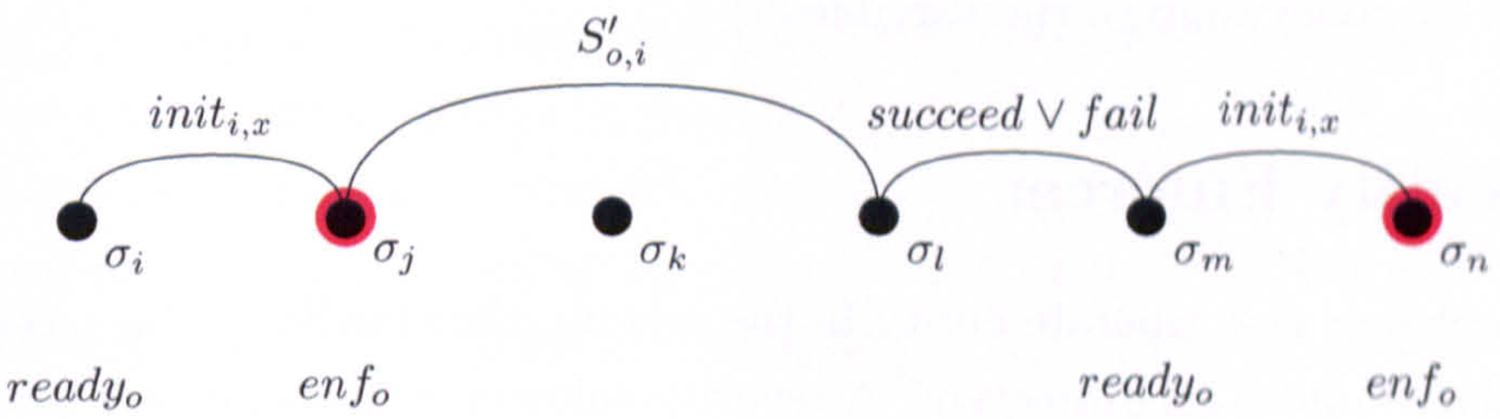


Figure 8.5: Abstraction Level for Environmental Integrity Policies

The object starts the execution of an interface in a state where it is *ready* (e.g. σ_i). It then initialises the local variables for the parameters. This is done in state σ_j. The marker enf_o is used to select this state in the abstraction for the enforcement of the policy. Following this, the interface executes on the local variables and either succeeds or fails. The interface invocation ends in the next state where the object is *ready* (σ_m). The results in the local variables remain stable from state σ_l to σ_m.

The problem is that the invocation of the next interface would potentially overwrite the local results and output parameters in the initialisation (state σ_m to σ_n). However, the policy semantics assumes that the results are available in the final state of the interval. To ensure that the results of the computation and the parameter values of the last execution are still available in state σ_n it is necessary to store them separately.

For all local variables v' that are assigned in the initialisation $init_{i,x}$ we introduce an auxiliary variable v'_h that reflects the value v' in the last interface execution. The refinement of the initialisation follows along the same lines as the refinement to introduce the marker variables.

$$init_{i,x} \mathrel{\widehat{=}} [\![\overline{v'} \leftarrow \overline{e}]\!]_{V^*}$$

Introduction of the new variables; Strengthening of the assignment and augmenting the control variables to ensure that the new variables are kept stable.

$$\sqsubseteq [\![\overline{v'}, \overline{v'_h} \leftarrow \overline{e}, \overline{v'}]\!]_{V^{1*}}$$

We denote here by V^* the original set of variables that are controlled by the initialisation and by V^{1*} the augmented set that contain the new variables. Using this two level approach, the results of the last execution are available in the next enforcement state σ_n. In more detail this means that the results have been computed in the local variables v' in state σ_l. They are then kept stable until state σ_m. Their values are then copied into the variables v'_h in state σ_n, viz. v'_h in σ_m has the same value as v' in σ_l. References to results in the integrity policy map to the variables v'_h.

8.7 Security Enforcer

A security enforcer is a separate entity in the system, that mediates the access to object interfaces placed under its protection. A security enforcer is like an object and can define additional interfaces and variables. In the following the set $O_{se,p}$ denotes the set of objects that are under the protection of the security enforcer *se*. $I_{se,p}$ denotes the set of interfaces that are protected by the security enforcer *se*. These are obtained from the **protect** clause in the specification of the security enforcer. The syntax allows to omit the second identifier of names. This means that all interfaces I_o of the object o are protected.

8.7.1 Enforceable Policies

The security enforcer can enforce environmental policies that control the protected interfaces. The security enforcer synchronises with all invocations of these interfaces and controls the access and integrity in a similar fashion as the vigilant object mechanism. The scope of the policies that can be maximally enforced by a security enforcer *se* is

$$(Subj_{se}, Obj_{se}, Act_{se}) = (\mathcal{A}, \{se\} \cup O_{se,p}, I_{se} \cup I_{se,p}\})$$

$\mathcal{A}$ is the set of all agents in the SMAS. $O_{se,p}$ is the set of all objects, that are listed in the **protect** part of *se*. I_{se} is the set of interfaces that are provided by *se* and $I_{se,p}$ is the set of interfaces that are listed in the **protect** part of *se*. For example the security enforcer definition:

```
securityenforcer se :
    protect (o1, o2.i1, o2.i2)
    var delegations = []
    delegate(in g, in d, in x, in y) { ... }
    revoke(in g, in d, in x, in y) { ... }
end
```

For this security enforcer, the scope of the enforceable policy is:

$$(\mathcal{A}, \{o_1, o_2, se\}, \{\texttt{delegate}, \texttt{revoke}, i_1, i_2\} \cup I_{o_1})$$

All subjects in the system can be controlled. Controlled objects are the security enforcer and the objects o_1 and o_2. Controlled interfaces are the interfaces that are defined by the enforcer (viz. `delegate` and `revoke`) and the interfaces under protection. The policies may reference the variables defined in the enforcer, but must not reference any variables defined in the objects under its protection. This means that the security enforcer observes the objects under its protection at the interface level. However, the policies may reference the parameters and control variables corresponding to the interfaces under protection.

8.7.2 Motivating Example

The system provides two objects `paypal` and `itune`. The object `paypal` provides the interface `pay(in amount)` that allows agents to make an on-line payment. The protection requirement for the objects is that whilst everybody can access the interface `pay` of `paypal`, only those agents that previously paid 5 (GBP) to `paypal` can access the interface `download(out mp3)` of object `itune`. This is a common scenario, e.g. a payment or non-disclosure statement must be made before the access is granted or a third party needs to give its consent.

Listing 8.1: Motivating Example for the Security Enforcer

```
object paypal:   /*...*/
  pay(in amount) { /* ... */  }
end

object itune :   /* ... */
  download(out mp3) { /* ... */  }
end

policy p : (
  allow (S,paypal,pay) when true
  allow (S,itune,download) when done(S,paypal,pay(x)) and x >= 5
  decide (S,O,A) when 0: allow(S,O,A)
) end

securityenforcer se : protect (paypal, itune) end
enforce p with se
```

The above policy would not be enforceable using any other mechanism because it does relate the execution of object o1 and o2. To ensure that the enforcer does not loose any policy related events, it serialises the access to all objects under its protection. This is explained in the next section.

8.7.3 Semantics of the Security Enforcer

The security enforcer is a special form of object, that mediates the access to the interfaces under its protection. The semantics of the security enforcer is an extension of the object semantics that was provided in Chapter 6.

They key difference is that the security enforcer can either participate in the execution of one of its own interfaces or in the execution of one of the interfaces it is protecting. The definition of the security enforcer is similar to the one for objects (see (6.13) on page 141).

$$\varphi_{se} \mathrel{\widehat{=}} init_{se} \,;\, ((\bigoplus_{x \in X_{se}} \psi_{se,x}) \oplus (\bigoplus_{x \in X_{se,p}} \psi_{o,x}))^* \tag{8.16}$$

Here X_{se} is the set of all remote actions that invoke one of the security enforcers interfaces. $X_{se,p}$ denotes the set of all remote actions that invoke one of the interfaces $I_{se,p}$ the security enforcer protects. The definitions of $init_{se}$ and $\psi_{se,x}$ are the same as the initialisation and interface invocation of an object (see (6.14), page 141 and (6.15), page 142).

To ensure that the security enforcer synchronises with the execution of the interfaces under its protection, the set of participating entities $E_{a,x}$ (see (6.2), page 135) is redefined for the SMAS_3 to include the security enforcer.

The entities that are involved in the execution of the action x by the agent a are

$$E_{a,x} = \begin{cases} \{a\} & \text{if } x \in X_a^{loc} \\ \{a, o\} \cup SE_x & \text{if } x \in X_a^{rmt} \end{cases} \tag{8.17}$$

Where the set SE_x is the set of all security enforcers that protect the interface i that the remote action x is invoking.

$$SE_x = \{se \mid se \in SE_{secenf} \wedge i \in I_{se,p}\} \tag{8.18}$$

This guarantees that all security enforcers synchronise with the execution of a remote action that is invoking an interface under their protection.

The restriction of having only one enforcer for per object means that the set contains either none or exactly one security enforcer. However, with a future extension of conflict

resolution between enforcers in place the set can be larger.

8.7.4 Mapping Policy- and System States

The abstraction for the security enforcer is the same as the abstraction for vigilant objects. However, the policy enforced by the security enforcer is interpreted over the states where the control variable enf_o of *any* of the objects $o \in Obj_{se}$ that are under the protection of the security enforcer se are *true*.

The synchronisation of the security enforcer, objects and invoking agents ensures that at most one object $o \in Obj_{se}$ can be accessed at any point in time. We define the S_{se} that bridges between the projected states as:

$$S_{se} \mathrel{\widehat{=}} \left(\bigvee_{o \in Obj_{se}} enf_o \wedge \mathsf{skip} \right) ; \left(\mathsf{keep} \bigwedge_{o \in Obj_{se}} \neg enf_o \right) \wedge \left(\mathsf{fin} \left(\bigvee_{o \in Obj_{se}} enf_o \right) \right) \tag{8.19}$$

This selects those states where any of the protected objects or the security enforcer itself have set the marker enf_o.

The enforcer enforces the complete policy P'_{se}:

$$P'_{se} \mathrel{\widehat{=}} Comp(Subj_{se}, Obj_{se}, Act_{se}, P_{se}) \tag{8.20}$$

The formula T_{se} that holds over the projected interval is the policy that is enforced in conjunction with the enforcement properties.

$$T_{se} \mathrel{\widehat{=}} P'_{se} \wedge EP_{se,autho} \wedge EP_{se,integ} \tag{8.21}$$

We augment definition (8.13) with the semantics of the security enforcer and the enforcement of policies to the final version of the $SMAS_3$ semantics:

$$SMAS_3 \mathrel{\widehat{=}} SMAS_2 \wedge bind \wedge \tag{8.22}$$
$$\bigwedge_{a \in SE_{vngagent}} (\mathsf{keep}\,(\neg\, ready_a))\,;\,(S_a \Delta T_a)) \wedge$$
$$\bigwedge_{o \in SE_{vngobject}} ((\mathsf{keep}\,(\neg\, ready_o))\,;\,(\mathsf{keep}\,(\neg\, enf_o))\,;\,(S_o \Delta T_o)) \wedge$$
$$\bigwedge_{se \in SE_{secenf}} \varphi_{se} \wedge$$
$$\bigwedge_{se \in SE_{secenf}} \left((\mathsf{keep}\,(\neg\, ready_{se}))\,;\,(\mathsf{keep}\,(\neg(\bigvee_{o \in Obj_{se}} enf_o)))\,;\,(S_{se} \Delta T_{se}) \right)$$

This means that the enforcement of a policy by a security enforcer mechanisms starts with the first interface invocation *after* the security enforcer has been initialised.

8.7.5 Enforcing Authorisation

The enforcement property for authorisation is similar to the one defined for the vigilant object mechanism. However, it applies to the maximal enforceable scope of the security enforcer.

$$EP_{se,autho} \mathrel{\widehat{=}} \bigwedge_{\substack{s \in Subj_{se} \\ o \in Obj_{se} \\ a \in Act_{se}}} \mathsf{keep}\,(\neg\, autho(s,o,a) \supset \bigcirc \neg\, done(s,o,a)) \tag{8.23}$$

8.7.6 Enforcing Integrity

The enforcement property for integrity is similar to the one defined for the vigilant object mechanism. However, it applies to the maximal enforceable scope of the security enforcer.

$$EP_{se,integ} \mathrel{\widehat{=}} \bigwedge_{\substack{s \in Subj_{se} \\ o \in Obj_{se} \\ a \in Act_{se}}} \Box(\neg\, integ(s,o,a) \supset \neg\, done(s,o,a)) \tag{8.24}$$

8.8 Summary

In this chapter we have shown what the enforcement of the different types of policies using the three enforcement mechanisms *vigilant agent*, *vigilant object* and *security enforcer* means and how policies can be associated with enforcement mechanisms in the SANTA language. Security Enforcers have been introduced as special objects that mediate the access to interfaces under their protection.

We defined for each enforcement mechanism the abstraction over which the enforcement of the policy is defined. The intuition is that the vigilant agent makes one policy decision before entering its execution phase. For vigilant objects the intuition is that the policy decision is made for every invocation. The security enforcer synchronises the access to all interfaces under its protection and defines the result of the policy decision for each interface invocation. It has been shown that these are suitable abstractions.

In addition to the abstraction level, the meaning of enforcing the different types of policies has been captured in enforcement properties. They define the relation between the object and agent control variables and the enforcement variables that reflect policy decisions. The abstraction and the enforcement properties provide the high-level specification of what enforcement means in the $SMAS_3$.

In the following chapter we consider the development that leads from the abstract specification of agents and objects and enforcement mechanisms to concrete implementable code.

Chapter 9

Refinement

This chapter provides refinement rules that are used for the stepwise refinement of the abstract specification constructs in SANTA into concrete, implementable constructs. The application of the rules is illustrated using the motivating example of Chapter 5 that is extended to enforce a simple policy vigilantly.

9.1 Introduction

Refinement is the provably correct transformation of an abstract specification into a concrete (deterministic) program. A good introduction to refinement is given in e.g. [96]; Back et.al. address the refinement of sequential and concurrent programs in [13, 11]; Refinement of ITL specifications into executable programs has been investigated in [41]. The specification-oriented semantics of the SANTA language allows to define transformations, so-called refinement rules, that can be applied to a specification to make it more deterministic. The iterative application of the rules is called *stepwise refinement*.

In SANTA some statements define the behaviour of the system at a more abstract level than others. The *temporal assignment* for example defines only the value of the variable in the final state of the interval — it does not define any value for intermediate states, nor the length of the interval over which the assignment takes place. In contrast a concrete assignment defines the length of the interval to be exactly one. Similarly the enforcement mechanisms define the effect that policy decisions have on the agents and objects in the SMAS at an abstract level. They must also be refined into concrete enforcement code that guarantees the compliance with the enforcement properties at the high-level.

In SANTA we use the notion of *delayed refinement*. By this we mean, that part of the system can be refined into concrete and implementable constructs, while another part remains at the abstract level. Delayed refinement allows to focus on one aspect of the system at a time. In SANTA delayed refinement is possible because of the compositional semantics of the underlying logic ITL and the fixed structure of the SMAS, i.e. the separation into agents and objects that encapsulate variables.

The chapter is structured as follows. In Section 9.2 we define the notion of refinement that is used in this work and prove some of the less obvious refinement rules that are used in the subsequent examples. In Section 9.3 we show how the different phases in the execution of the SANTA agent that was described informally in Section 5.2.4 can be refined into deterministic ones. This shows how the refinement rules are applied to develop a concrete and implementable agent. Subsequently in Section 9.4 we show how policies that are enforced vigilantly by an agent are refined into concrete enforcement code. The advantage of this approach is that it allows to give guarantees on the time that the enforcement mechanisms need to compute the policy decision in a state. The potential benefits are similar to those presented in [111, 118], however we have a sound mathematical foundation that allows to reason about and proof timing properties. We conclude the chapter in Section 9.5 with a short summary.

9.2 Refinement Rules

Refinement in this work means implication. A specification $spec_1$ is refined by another specification $spec_2$ iff $spec_2 \supset spec_1$.

$$spec_1 \sqsubseteq spec_2 \mathrel{\widehat{=}} spec_2 \supset spec_1$$

The specification oriented semantics of SANTA makes it possible to prove that one SANTA program is a refinement of another. This is used to gradually refine the abstract specification of the SANTA program towards a program that contains only implementable constructs, viz. the timing of statements is deterministic and enforcement mechanisms are implemented to ensure the compliance with policies and enforcement properties. The following theorems are not a complete list of all refinement rules, but rather a collection of the less obvious refinements used in the examples.

Theorem 7 (Temporal to Concrete Parallel Assignment)
The abstract multiple assignment is refined by the concrete multiple assignment.

$$[\![x_1 \ldots x_n \leftarrow e_1 \ldots e_n]\!]_V \sqsubseteq [\![x_1 \ldots x_n := e_1 \ldots e_n]\!]_V$$

Proof 7 (Temporal to Concrete Assignment)
The abstract multiple assignment with respect to to a set of variables V is defined as:

$$[\![\boxed{x_0, \ldots, x_n \leftarrow e_0, \ldots, e_n}]\!]_V \mathrel{\widehat{=}} \bigwedge_{0 \le i \le n} (x_i \leftarrow e_i) \wedge \bigwedge_{u \in V \setminus \{x_0, \ldots, x_n\}} \mathsf{stable}(u)$$

the concrete multiple assignment as:

$$[\![\boxed{x_0, \ldots, x_n := e_0, \ldots, e_n}]\!]_V \mathrel{\widehat{=}} \mathsf{skip} \wedge \bigwedge_{0 \le i \le n} ((\bigcirc x_i) = e_i) \wedge \bigwedge_{u \in V \setminus \{x_0, \ldots, x_n\}} \mathsf{stable}(u)$$

In both cases the last conjunct keeps all variables in V that are not assigned, stable throughout the interval. It remains to be shown that $x := e$ is a refinement of $x \leftarrow e$.

$x \leftarrow e \mathrel{\widehat{=}} \mathsf{finite} \wedge \mathsf{fin}(x) = e$	*by definition*
$\sqsubseteq \mathsf{skip} \wedge \mathsf{fin}(x) = e$	*because:* $\mathsf{skip} \supset \mathsf{finite}$
$\sqsubseteq \mathsf{skip} \wedge (\bigcirc x) = e$	*because:* $\mathsf{skip} \supset ((\bigcirc x) \equiv \mathsf{fin}(x))$
$\mathrel{\widehat{=}} x := e$	*by definition*

The final state in a unit interval is identical to the next state.

Theorem 8 (Prefix Introduction for Temporal Multiple Assignment)
An temporal multiple assignment can be refined into an idle prefix followed by the temporal multiple assignment.

$$[x \leftarrow e]_V \sqsubseteq [\mathsf{idle}; x \leftarrow e]_V$$

provided $x \in V$ and that e is an expression on variables in V.

Proof 8 (Prefix Introduction for Temporal Multiple Assignment)
Proof that

$$[\mathsf{idle}; x \leftarrow e]_V \supset [x \leftarrow e]_V$$

Assumptions: $x \in V$ *and* $(\bigwedge_{v \in V} \mathsf{stable}\ v) \supset \mathsf{stable}\ e$.

$$[x \leftarrow e]_V \mathrel{\widehat{=}} (x \leftarrow e \wedge \bigwedge_{v \in V \setminus \{x\}} \mathsf{stable}\ v) \tag{9.1}$$

Introduction of a unit interval that keeps the expression e stable. Using the assumption all variables in V are kept stable. The length of the interval is still finite: (skip;finite) $\supset$ finite.

$$\sqsubseteq ((\mathsf{skip} \wedge \bigwedge_{v \in V} \mathsf{stable}\ v)\,; x \leftarrow e) \wedge \bigwedge_{v \in V \setminus \{x\}} \mathsf{stable}\ v) \tag{9.2}$$

Importing the conjunct $(\bigwedge_{v \in V \setminus \{x\}} \mathsf{stable}\ v)$ *in the Chop yields:*

$$\equiv ((\mathsf{skip} \wedge \bigwedge_{v \in V} \mathsf{stable}\ v))\,; ((x \leftarrow e) \wedge \bigwedge_{v \in V \setminus \{x\}} \mathsf{stable}\ v) \tag{9.3}$$

The right hand side of the Chop is the definition of the temporal parallel assignment. The left hand side of the Chop is the definition of idle.

$$\equiv [\mathsf{idle}]_V\,;[x \leftarrow e]_V \tag{9.4}$$

Using the statement semantics of the sequential composition this can be combined to:

$$\equiv [\mathsf{idle}; x \leftarrow e]_V \tag{9.5}$$

Theorem 9 (Extended Prefix for Temporal Multiple Assignment)
From the iterative application of Theorem 8 follows that:

$$[x \leftarrow e]_V \sqsubseteq (([\mathsf{idle}]_V)^* \wedge \mathsf{finite})\,;[x \leftarrow e]_V$$

Proof 9 (Extended Prefix for Temporal Multiple Assignment)
The proof is straightforward from the iterative application of Theorem 8. Restricting the Chopstar to a finite interval is necessary.

Theorem 10 (Sequential Decomposition of Temporal Multiple Assignment)
An temporal multiple assignment can be decomposed sequentially.

$$\begin{aligned}[\![x_1 \ldots x_n \leftarrow e_1 \ldots e_n]\!]_V \sqsubseteq [\![& \textsf{var } t_1 \ldots t_j \leftarrow x_1 \ldots x_j \ : \{ \\ & x_1 \ldots x_j \leftarrow e_1 \ldots e_j \ ; \\ & x_{j+1} \ldots x_n \leftarrow e'_{j+1} \ldots e'_n \\ & \}]\!]_V\end{aligned}$$

where $1 \leq j \leq n$ *and* $1 \leq i \leq j$ *and* $e'_i \mathrel{\widehat{=}} e_i[t_1 \ldots t_j / x_1 \ldots x_j]$ *is the expression* e_i *where every occurrence of* $x_1 \ldots x_j$ *is substituted with the local variable* $t_1 \ldots t_j$.

Proof 10 (Sequential Decomposition of Temporal Multiple Assignment)
The introduction of local variables is a refinement.

$$\begin{aligned}&[\![x_1 \ldots x_n \leftarrow e_1 \ldots e_n]\!]_V \\ &\sqsubseteq \exists t_1 \ldots t_j \cdot ([\![x_1 \ldots x_n \leftarrow e_1 \ldots e_n]\!]_V)\end{aligned}$$

The application of Theorem 9 yields:

$$\sqsubseteq \exists t_1 \ldots t_j \cdot (\textsf{finite} \wedge ([\![\textsf{idle}]\!]_V)^* \ ; [\![x_1 \ldots x_n \leftarrow e_1 \ldots e_n]\!]_V)$$

The left hand side of the Chop can be refined by an temporal assignment that maintains all variables in V. *Additionally the set of variables* V *is extended by the set of local variables, to keep them stable.*

$$\sqsubseteq \exists t_1 \ldots t_j \cdot ([\![t_1 \ldots t_j \leftarrow x_1 \ldots x_j]\!]_{V \cup \{t_1 \ldots t_j\}} \ ; [\![x_1 \ldots x_j \ldots x_n \leftarrow e_1 \ldots e_j \ldots e_n]\!]_{V \cup \{t_1 \ldots t_j\}})$$

The temporal assignment keeps all variables in V *stable over the finite interval in which the local variables* $t_1 \ldots t_j$ *are assigned. In the final state of the assignment the local variables have the same value as the corresponding variables* $x_1 \ldots x_j$. *This is used to substitute all occurrences of* $x_1 \ldots x_j$ *in the expressions* $e_{j+1} \ldots e_n$ *with the local variables* $t_1 \ldots t_j$. *The expression after the substitution is denoted by* $e'_i = e_i[t_1 \ldots t_j / x_1 \ldots x_j], j < i \leq n$.

$$\begin{aligned}\sqsubseteq \exists t_1 \ldots t_j \cdot (& [\![t_1 \ldots t_j \leftarrow x_1 \ldots x_j]\!]_{V \cup \{t_1 \ldots t_j\}}); \\ & [\![x_1 \ldots x_j, x_{j+1} \ldots x_n \leftarrow e_1 \ldots e_j, e'_{j+1} \ldots e'_n]\!]_{V \cup \{t_1 \ldots t_j\}}\end{aligned}$$

Splitting the assignment into a prefix and a suffix:

$$\sqsubseteq \exists t_1 \ldots t_j \cdot ([\![t_1 \ldots t_j \leftarrow x_1 \ldots x_j]\!]_{V \cup \{t_1 \ldots t_j\}});$$
$$[\![x_1 \ldots x_j \leftarrow e_1 \ldots e_j \; ; \; x_{j+1} \ldots x_n \leftarrow e'_{j+1} \ldots e'_n]\!]_{V \cup \{t_1 \ldots t_j\}}$$

is a refinement, because the values that have been assigned to $x_1 \ldots x_j$ in the prefix are kept stable over the suffix, and the expressions $e'_{j+1} \ldots e'_n$ do not depend on $x_1 \ldots x_j$. The substitution with the local variables ensured that the expressions $e'_{j+1} \ldots e'_n$ yield the same result as in the expressions $e_{j+1} \ldots e_n$ in the initial state.

$$\sqsubseteq \exists t_1 \ldots t_j \cdot ([\![t_1 \ldots t_j \leftarrow x_1 \ldots x_j]\!]_{V \cup \{t_1 \ldots t_j\}});$$
$$[\![x_1 \ldots x_j \leftarrow e_1 \ldots e_j \; ; \; x_{j+1} \ldots x_n \leftarrow e'_{j+1} \ldots e'_n]\!]_{V \cup \{t_1 \ldots t_j\}}$$

Using the semantics of the sequential composition and the semantics of local variable introduction, this can be rewritten to:

$$\equiv [\![\textsf{var } t_1 \ldots t_j \leftarrow x_1 \ldots x_j \; : \{x_1 \ldots x_j \leftarrow e_1 \ldots e_j \; ; \; x_{j+1} \ldots x_n \leftarrow e'_{j+1} \ldots e'_n\}]\!]_V$$

Theorem 11 (Remove unreferenced Local Variables)
A local variable t_1 that is never referenced in stat can be removed.

$$[\![\textsf{var } t_1, \ldots, t_n \leftarrow expr_1, \ldots, expr_n : stat]\!] \sqsubseteq [\![\textsf{var } t_2, \ldots, t_n \leftarrow expr_2, \ldots, expr_n : stat]\!]$$

If t_1 is the only local variable, the var *construct can be removed.*

9.3 Refinement of the $SMAS_1$ Example

When refining the example all temporal multiple assignments must be expressed as concrete multiple assignments or sequences of concrete assignments. The specification level behaviour of the example is included here again, for the readers convenience.

```
1  agent ag :
2      var x = 0
3      when x < 2^32 do inc : x := x+1
4      when x < 2^31 do dbl : x := 2*x
5
6      deliberation : {
7      if x % 2 = 0 then inc,dbl ← 1,0
8                   else inc,dbl ← 0,1 }
9  end
```

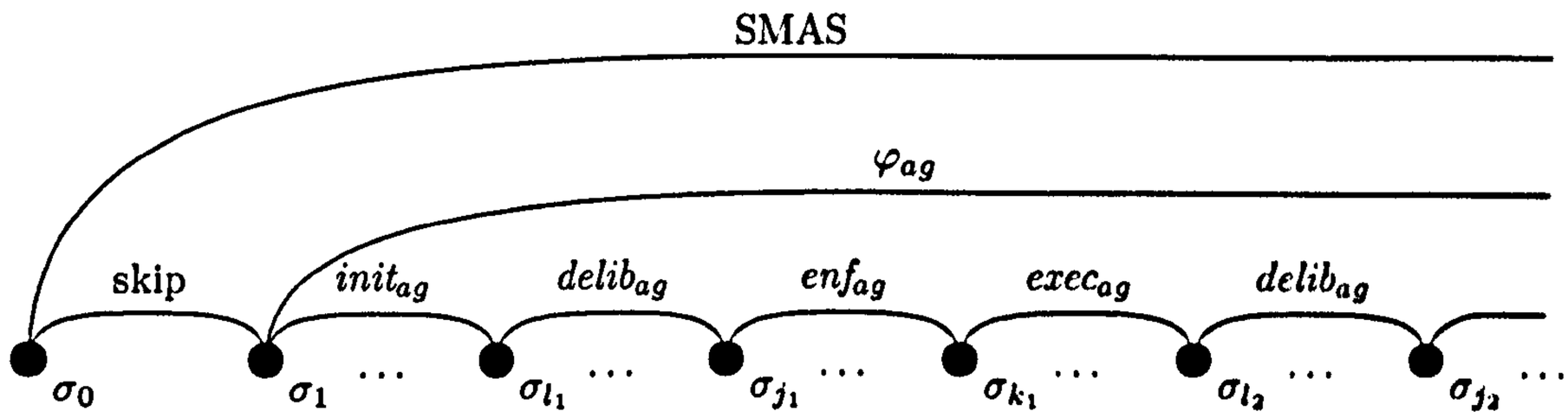


Figure 9.1: Specification level behaviour of the SMAS$_1$ example

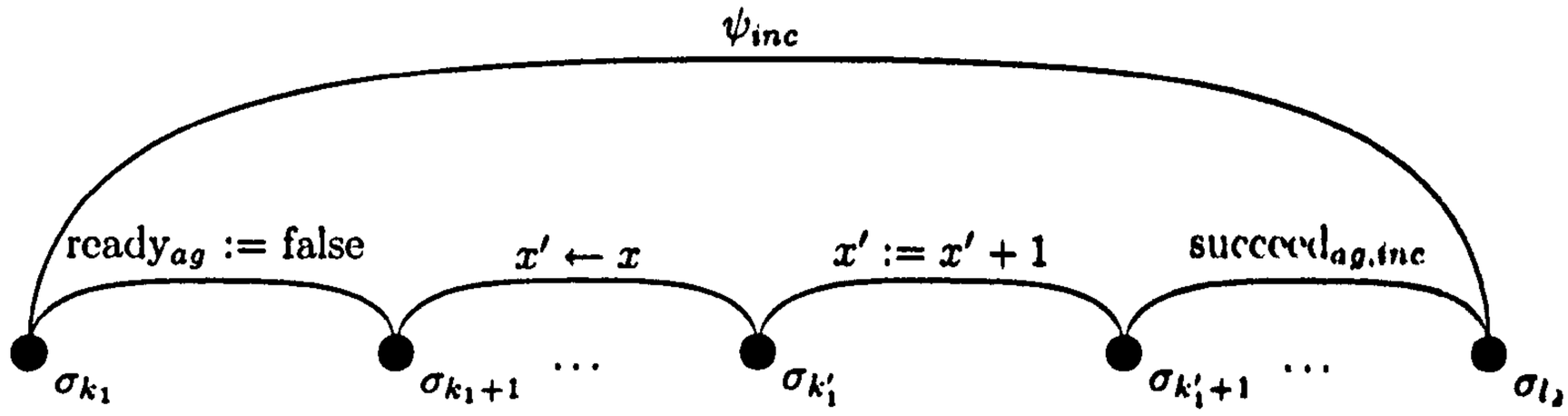


Figure 9.2: Specification level behaviour of action `inc`

9.3.1 Initialisation

The semantics of the initialisation phase states that the control variable $ready_{ag}$ is *false* throughout the initialisation, and that all other control variables are assigned to their default value at the end of the initialisation phase. Additionally all agent variables are assigned their initial value.

All the agent state variables V_{ag} are known. The specification of the initialisation phase (see Section 5.3.1, Equation (5.1)) can be expressed as the following SANTA statement.

$$init_{ag} \mathrel{\widehat{=}} ready_{ag} = false \wedge \llbracket doterm_{ag}, term_{ag}, done_{ag,inc}, done_{ag,dbl}, failed_{ag,inc}, failed_{ag,dbl}, \Pi_{ag,inc}, \Pi_{ag,dbl}, x \leftarrow false, false, false, false, false, false, 1, 1, 0 \rrbracket_{V_{ag}}$$

It is assumed, that the control variable $ready_a$ is correctly initialised to *false*. This statement is clearly satisfying the specification, as it guarantees that the variable $ready_{ag}$ is kept stable (viz. *false*) throughout the duration of the temporal assignment. The temporal multiple assignment can then be refined using the refinement rules that have been provided in the previous section. First Theorem 10 is applied to split the assignment:

$$init_{ag} \sqsubseteq ready_{ag} = false \wedge$$
$$[\textbf{var } t_1 \leftarrow doterm_{ag} : \{$$
$$doterm_{ag} \leftarrow false;$$
$$term_{ag}, done_{ag,inc}, done_{ag,dbl}, failed_{ag,inc}, failed_{ag,dbl}, \Pi_{ag,inc}, \Pi_{ag,dbl}, \mathtt{x} \leftarrow$$
$$false, false, false, false, false, 1, 1, 0$$
$$]v_{ag}$$

The local variable t_1 is never referenced in the following statement. The application of Theorem 11 removes this unreferenced variable:

$$init_{ag} \sqsubseteq ready_{ag} = false \wedge$$
$$[doterm_{ag} \leftarrow false;$$
$$term_{ag}, done_{ag,inc}, done_{ag,dbl}, failed_{ag,inc}, failed_{ag,dbl}, \Pi_{ag,inc}, \Pi_{ag,dbl}, x \leftarrow$$
$$false, false, false, false, false, 1, 1, 0$$
$$]v_{ag}$$

The repeated application of Theorems 10 and 11 yields:

$$init_{ag} \sqsubseteq ready_{ag} = false \wedge$$
$$[doterm_{ag} \leftarrow false\ ;\ term_{ag} \leftarrow false;$$
$$done_{ag,inc}, done_{ag,dbl}, failed_{ag,inc}, failed_{ag,dbl} \leftarrow false, false, false, false;$$
$$\Pi_{ag,inc} \leftarrow 1\ ;\ \Pi_{ag,dbl} \leftarrow 1\ ;\ \mathtt{x} \leftarrow 0$$
$$]v_{ag}$$

Using Theorem 7 the temporal multiple assignment is refined to the concrete (multiple)

assignment.

$$
\begin{aligned}
init_{\mathtt{ag}} \sqsubseteq\ & ready_{\mathtt{ag}} = false \wedge \\
& [\![doterm_{\mathtt{ag}} := false \,;\, term_{\mathtt{ag}} := false; \\
& \quad done_{\mathtt{ag,inc}}, done_{\mathtt{ag,dbl}}, failed_{\mathtt{ag,inc}}, failed_{\mathtt{ag,dbl}} := false, false, false, false; \\
& \quad \Pi_{\mathtt{ag,inc}} := 1 \,;\, \Pi_{\mathtt{ag,dbl}} := 1 \,;\, x := 0 \\
&]\!]_{V_{\mathtt{ag}}}
\end{aligned}
$$

It is assumed, that the boolean control variables indicating the success or failure of the previously executed action are implemented as an integer, that represents the four boolean variables and allows for their parallel assignment. All assignments are now concrete and the length of the initialisation phase is determined. This is depicted in Figure 9.3

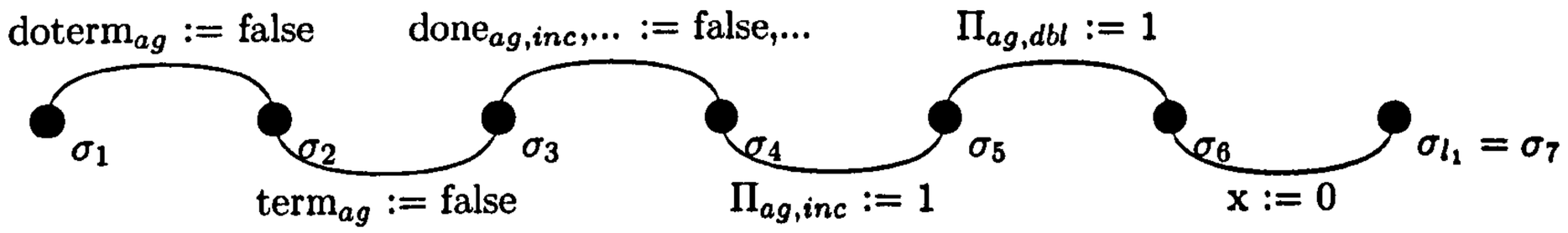


Figure 9.3: Implementation Level behaviour of the initialisation phase

The states are listed in the table below:

Agent Variables		σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	$\sigma_{l_1=7}$
x	Variable x	?	?	?	?	?	?	0
Priority Variables								
$\Pi_{\mathtt{ag},inc}$	Priority of `inc`	?	?	?	?	1	1	1
$\Pi_{\mathtt{ag},dbl}$	Priority of `dbl`	?	?	?	?	?	1	1
Control Variables								
$ready_{\mathtt{ag}}$	ag is *ready*	F	F	F	F	F	F	F
$done_{\mathtt{ag,inc}}$	ag successfully executed `inc`	?	?	?	F	F	F	F
$failed_{\mathtt{ag,inc}}$	ag failed executing `inc`	?	?	?	F	F	F	F
$done_{\mathtt{ag,dbl}}$	ag successfully executed `dbl`	?	?	?	F	F	F	F
$failed_{\mathtt{ag,dbl}}$	ag failed executing `dbl`	?	?	?	F	F	F	F
$term_{\mathtt{ag}}$	Final state of the DEE cycle	?	?	F	F	F	F	F
$doterm_{\mathtt{ag}}$	Indicate termination at the end of DEE	?	F	F	F	F	F	F

Table 9.1: States in the Initialisation Phase

The temporal multiple assignment has been refined into a sequence of assignments of length 6. The refined initialisation phase contains now only concrete assignments and its implementation is straight forward. Other refinements are obviously possible.

9.3.2 Deliberation

The deliberation phase is refined in a similar fashion. The statement is explicitly defined in the deliberation section and assigns the priority variables to non-negative values, while maintaining all other agent state variables.

$$
\begin{aligned}
delib_{ag,explicit} &\widehat{=} [\![S]\!]_{V_{ag}} \wedge delib_{ag,external} \\
&\equiv [\![\boxed{\texttt{if x\%2=0 then inc,dbl}\leftarrow \texttt{1,0 else inc,dbl}\leftarrow \texttt{0,1}}]\!]_{V_{ag}} \wedge delib_{ag,external}
\end{aligned}
$$

The statement clearly implies $delib_{ag,external}$, as only priority variables are modified and assigned to non-negative values.

$$
\sqsubseteq [\![\boxed{\texttt{if x\%2=0 then inc,dbl}\leftarrow \texttt{1,0 else inc,dbl}\leftarrow \texttt{0,1}}]\!]_{V_{ag}}
$$

The application of Theorems 10 and 11 yields then:

$$
\sqsubseteq [\![\boxed{\texttt{if x\%2=0 then \{inc:=1;dbl:=0\}else \{inc:=0;dbl:=1\}}}]\!]_{V_{ag}}
$$

From the initialisation it is clear that in the first state σ_7 the value of x is 0. This means that in the first iteration of the DEE cycle the *true* branch of the conditional is executed. This leads to the following behaviour:

Agent Variables		σ_8	$\sigma_{j_1} = \sigma_9$	$\sigma_{k_1} = \sigma_{10}$
x	Variable x	0	0	0
Priority Variables				
$\Pi_{ag,inc}$	Priority of inc	1	1	1
$\Pi_{ag,dbl}$	Priority of dbl	1	0	0
Control Variables				
$ready_{ag}$	ag is *ready*	F	F	T
$done_{ag,inc}$	ag successfully executed inc	F	F	F
$failed_{ag,inc}$	ag failed executing inc	F	F	F
$done_{ag,dbl}$	ag successfully executed dbl	F	F	F
$failed_{ag,dbl}$	ag failed executing dbl	F	F	F
$term_{ag}$	Final state of the DEE cycle	F	F	F
$doterm_{ag}$	Indicate termination at the end of DEE	F	F	F

Table 9.2: States in the Deliberation and Enforcement Phase

9.3.3 Enforcement

Since no policies are defined to be vigilantly enforced by the agent, the enforcement phase does not affect the priority decisions made in the deliberation step.

$$enf_{ag} \mathrel{\widehat{=}} \left(\mathsf{finite} \wedge \bigwedge_{x \in X_{ag}} \mathsf{fin}\,(\top_\Pi \leq \Pi_{ag,x} \leq \bot_\Pi) \wedge \bigwedge_{v \in V} (\mathsf{keep\ stable}\ v) \right) ; [\![\boxed{ready_{ag} := true}]\!]_{V_{ag}}$$

The initial state always satisfies $\bigwedge_{x \in X_{ag}} 0 \leq \Pi_{ag,x} \leq \bot_\Pi$ (see deliberation), therefore the empty interval is a trivial refinement of the above prefix. (empty $\supset$ finite and empty $\supset$ keep f).

$$\sqsubseteq \mathsf{empty} ; [\![\boxed{ready_{ag} := true}]\!]_{V_{ag}} \equiv [\![\boxed{ready_{ag} := true}]\!]_{V_{ag}}$$

The empty prefix can be omitted. No policy is enforced in this phase and the agent can immediately enter the execution phase.

9.3.4 Execution

In the execution phase the agent chooses one of the enabled actions non-deterministically or remains *idle*. An action is enabled, if the agent is *ready*, its precondition and its guard evaluate to *true*. The chosen action is then executed on temporary variables and either succeeds or fails.

The choice between success or failure is non-deterministic in the specification and refined into a conditional choice that reflects the outcome of an integrity check. As no policies are being enforced, the action can only succeed. This means that the statement execution for the action `inc` (see Equation (5.16)):

$$stat_{ag,inc} \mathrel{\widehat{=}} \exists x' \cdot ([\![\boxed{x' \leftarrow\ x}]\!]_{V_{ag}^+} ; [\![\boxed{\texttt{x':=x'+1}}]\!]_{V_{ag}^+} ; (succeed_{ag,inc} \oplus fail_{ag,inc}))$$

is refined to:

$$\begin{aligned}
&\sqsubseteq \exists x' \cdot ([\![\boxed{x' \leftarrow\ x}]\!]_{V_{ag}^+} ; [\![\boxed{\texttt{x':=x'+1}}]\!]_{V_{ag}^+} ; succeed_{ag,inc}) \\
&\equiv \exists x' \cdot ([\![\boxed{x' \leftarrow\ x}]\!]_{V_{ag}^+} ; [\![\boxed{\texttt{x':=x'+1}}]\!]_{V_{ag}^+} ; \\
&[\![\boxed{done_{ag,inc}, done_{ag,dbl}, failed_{ag,inc}, failed_{ag,dbl}, x \leftarrow\ \texttt{true,false,false,false,}\ x'}]\!]_{V_{ag}^+})
\end{aligned}$$

Given the semantics of the sequential composition of statements this can be written as:

$$\equiv \exists x' \cdot ([x' \leftarrow x; x' := x' + 1;$$
$$done_{ag,inc}, done_{ag,dbl}, failed_{ag,inc}, failed_{ag,dbl}, x \leftarrow \texttt{true,false,false,false}, x']_{V_{ag}^+})$$

With the definition of local variables (see Equation (5.10)) this can be rewritten to:

$$\equiv [\textsf{var } x' \leftarrow x: \{ x' := x' + 1;$$
$$done_{ag,inc}, done_{ag,dbl}, failed_{ag,inc}, failed_{ag,dbl}, x \leftarrow \texttt{true,false,false,false}, x'\}]_{V_{ag}}$$

The temporal multiple assignment can now be refined using the Theorems 10 and 11. Similarily the statement for the action `dbl` can be refined. The non-deterministic choice between the enabled actions can be refined to a conditional choice. In this example this is straight-forward.

$$\varphi_{ag} \mathrel{\widehat{=}} g_{ag,inc} \wedge ready_{ag} \wedge ([ready_{ag} := false]_{V_{ag}} ; stat_{ag,inc}) \oplus$$
$$g_{ag,dbl} \wedge ready_{ag} \wedge ([ready_{ag} := false]_{V_{ag}} ; stat_{ag,dbl}) \oplus$$
$$idle_{ag}$$

Using the statement definition and the definition of the conditional choice we can refine the nondeterministic choice as:

$$\sqsubseteq [\textsf{if } (g_{ag,inc} \textsf{ and } ready_{ag}) \textsf{ then } ready_{ag} := \texttt{false} ; stat_{ag,inc}$$
$$\textsf{else if } (g_{ag,dbl} \textsf{ and } ready_{ag}) \textsf{ then } ready_{ag} := \texttt{false} ; stat_{ag,dbl}$$
$$\textsf{else } idle_{ag}]_{V_{ag}}$$

In other settings however, care must be taken to ensure that the fairness criterion is met. An implementation could for example randomly choose between enabled actions or implement more sophisticated scheduling algorithms. Table 9.3 shows the behaviour of the agent variables, assuming that the temporal multiple assignments are refined as shown in the earlier examples. We assume that the creation and assignment of the local variable x' takes one unit interval.

Agent Variables		σ_{11}	$\sigma_{k'_1=12}$	σ_{13}	σ_{14}	$\sigma_{l_2=15}$
x	Variable x	0	0	0	0	1
x'	Local Variable x'		0	1	1	1
Priority Variables						
$\Pi_{ag,inc}$	Priority of `inc`	1	1	1	1	1
$\Pi_{ag,dbl}$	Priority of `dbl`	0	0	0	0	0
Control Variables						
$ready_{ag}$	ag is *ready*	F	F	F	F	F
$done_{ag,inc}$	ag successfully executed `inc`	F	F	F	T	T
$failed_{ag,inc}$	ag failed executing `inc`	F	F	F	F	F
$done_{ag,dbl}$	ag successfully executed `dbl`	F	F	F	F	F
$failed_{ag,dbl}$	ag failed executing `dbl`	F	F	F	F	F
$term_{ag}$	Final state of the DEE cycle	F	F	F	F	F
$doterm_{ag}$	Indicate termination at the end of DEE	F	F	F	F	F

Table 9.3: States in the Execution Phase

This example shows that it is possible to refine a SANTA program from its abstract, specification oriented semantics into concrete, implementable constructs. At the concrete level all choices are deterministic, and an execution trace can be derived from the semantics. The refinement of the $SMAS_2$ is very similar to the refinement that was shown in this section and is not detailed.

9.4 Refining a Vigilant Agent

The previous section has shown how the abstract specification of a SANTA agent can be refined to be deterministic and implementable. The example assumed, that no enforcement mechanisms are in place and that any action execution does therefore succeed. In this section we concern ourselves with the refinement of a vigilant agent, that is enforcing a simple policy and show how the action selection and the non-deterministic choice between action success and failure is refined to ensure that the vigilant agent correctly enforces the policy, viz. satisfies the enforcement properties. This can be easily extended for the enforcement of composed policies.

To keep the example simple, we extend the definition of the $SMAS_1$ example that was refined in the previous section with a policy that is vigilantly enforced by the agent.

```
1  agent ag :
2      var x = 0
3      when x < 2^32 do inc : x := x+1
4      when x < 2^31 do dbl : x := 2*x
5
6      deliberation : {
7      if x % 2 = 0 then inc,DB← 1,0
8                   else inc,dbl← 0,1 }
9  end
10
11 policy p :
12 (   allow      (ag,ag,A)    when true ,
13     deny       (ag,ag,dbl)  when 1: (x < 5),
14     decide     (ag,ag,A)    when 0: ( allow(ag,ag,A) and
15                                       not deny(ag,ag,A)),
16     oblige     (ag,ag,inc)  when [T..2,2]: sometime (x < 3),
17     integrity  (ag,ag,A)    when 0: x' < 20
18 )
19 end
20
21 enforce p with ag
```

The agent definition remains the same as previously described. The agent would start the execution with the action `inc` and alternate continuously between the execution of `dbl` and `inc` until the value exceeds the integer limit. However, now the policy `p` that is enforced vigilantly by the agent `ag` does impose additional constraints.

The example of the agent has been deliberately kept simple. The complexity of the actual computation that is performed by the actions, or the complexity of the decision making is irrelevant for the enforcement of policies. The policy is defined at an abstract level where the details of the computation are hidden. At this level of abstraction it does not make a difference whether the action merely increments an integer or performs another complex computation. Similarly the complexity that is involved in the agent deliberation is not visible at this level — only the result, viz. the assignment of the priority values is observable.

The policy `p` shows an example for each type of rule and contains some rules that reference the history of the execution. The informal meaning of the rules is as follows:

Positive Authorisation The agent `ag` is unconditionally (viz. the premise of the rule is *true*) allowed to execute actions.

Negative Authorisation The agent `ag` is denied to perform the action `dbl` if the value of its agent variable `x` was in the previous state less than 5.

Decision Rule The agent `ag` may execute any of its actions provided that it is allowed (positive authorisation) and not denied (negative authorisation) to do so.

Obligation Rule The agent `ag` is obliged to execute the action `inc` if the value of `x` has been less than 3 at some point within the past interval of length 2. The interpretation of the past interval is lenient, viz. if there is not enough history available for the evaluation of the rule, the longest available past interval is used.

Integrity Rule Any action execution is considered to be failed if it would assign a value to `x` that is greater than 20.

These additional constraints on the execution *change* the actual execution of the agent quite dramatically. The result is that the agent will increment the value of `x` until it reaches 5 and remain *idle* for one step. Following this, the agent's execution is not affected by the policy and it alternates between doubling and incrementing the value of `x`. However, it will soon exceed the maximal value of 20 that is defined by the integrity rule and as a result constantly try to execute the action and fail. This is detailed in the following.

To enforce authorisation rules and obligation rules we implement the enforcement phase of the agent in such a way that the assignment of priority variables guarantees that none of the enforcement properties is violated. Table 9.4 shows the execution of the agent at the states where the agent is *ready*. To give a better intuition we provided two values for the priority variables. The first value denotes the outcome of the agent's deliberation. The second value is the value that is assigned during the enforcement phase of the agent, viz. that overrules the initial decision. If only one value is specified, it means that the enforcement phase does not modify the decision.

State	σ'_0	σ'_1	σ'_2	σ'_3	σ'_4	σ'_5	σ'_6	σ'_7	σ'_8	σ'_9
x =	0	1	2	3	4	5	5	10	11	11
Π_{inc}	1/-1	0/-1	1/-1	0/-1	1/-1	0/	0/	1/	0/	0/
Π_{dbl}	0/	1/0	0/0	1/0	0/0	1/0	1/	0/	1/	1/
$done_{ag,inc}$	F	T	T	T	T	T	F	F	T	F
$failed_{ag,inc}$	F	F	F	F	F	F	F	F	F	F
$done_{ag,dbl}$	F	F	F	F	F	F	F	T	F	F
$failed_{ag,dbl}$	F	F	F	F	F	F	F	F	F	T
$autho^+(ag, ag, inc)$	T	T	T	T	T	T	T	T	T	T
$autho^+(ag, ag, dbl)$	T	T	T	T	T	T	T	T	T	T
$autho^-(ag, ag, inc)$	F	F	F	F	F	F	F	F	F	F
$autho^-(ag, ag, dbl)$	F	T	T	T	T	T	F	F	F	F
$autho(ag, ag, inc)$	T	T	T	T	T	T	T	T	T	T
$autho(ag, ag, dbl)$	T	F	F	F	F	F	T	T	T	T
$oblig(ag, ag, inc)$	T	T	T	T	T	F	F	F	F	F
$oblig(ag, ag, dbl)$	F	F	F	F	F	F	F	F	F	F
$integ(ag, ag, inc)$	F	T	T	T	T	T	T	T	T	F
$integ(ag, ag, dbl)$	F	T	T	T	T	T	T	T	T	F

Table 9.4: Vigilantly Enforced Policy

In state σ'_0 the value of x is 0, the agent's deliberation phase decides on the execution of action `inc`. The evaluation of the policy shows that the agent is indeed allowed to execute any action. Since x is less than 3, the obligation rule fires ($\tau = 0$, viz. for the empty interval it is sometimes the case that x is less than 3). The integrity cannot be met in the inital state, as any integrity rule demands at least a history of one. This does not violate the enforcement property for integrity, because in the initial state it is also not possible that an action has been executed. To enforce the obligation, the priority Π_{inc} is assigned to the value -1, guaranteeing the execution of action `inc`.

In the next state (σ'_1) the agent has successfully executed the action `inc`, since the updated state met the integrity constraint ($x < 20$). The value of x is now 1. Consequently the agent decides to execute the action `dbl`. Evaluating the policy results in the assignment of Π_{inc} to -1 to meet the agent's obligation. Additionally the negative authorisation (x in the last state is less than 5) to execute action `dbl` means that Π_{dbl} is assigned to the value 0, guaranteeing that it is not executed. The agent executes the action `inc` again.

This continues until state σ'_5, in which the value of x has reached 5. In this state the agent is no longer obliged to execute `inc`, because there is no state in the past interval of length 2 where x is less than 3. Still the last value of x is less than 5, leading to a negative authorisation to execute action `dbl`. The agent's deliberation phase decided to execute

action `dbl`, which is overruled by the negative authorisation, and not to execute action `inc`. The agent remains *idle*, because no action can be chosen for execution. This is reflected in the assignment of the control variables *done* and *failed* in the next state — they are all *false*.

The case for σ'_6 is similar to the previous one, however, the negative authorisation cannot be derived. This means that the agent does not execute the action `dbl` and thus doubled the value of `x` from 5 to 10. From now on the agent behaves *normally*, viz. the decision made by its deliberation phase is not overruled in the enforcement phase. All actions executions are permissible and no obligations can be derived. This continues until state σ'_8.

In state σ'_8 the agent decides on doubling the value of `x` from 11 to 22. The execution of the action `dbl` starts, but the result is rejected by the integrity policy. This means that the value of `x` remains 11 in state σ'_9. The failure to execute is indicated in the control variable $failed_{ag,dbl}$. Unfortunately the deliberation of the agent is not very sophisticated and does not take into account the possibility for failure. Consequently the same decision to execute the action `dbl` will be made again. This continues indefinitely.

In the above example the policy decisions have been made by interpreting the semantics of the policy rules. However the aim of this chapter is to show how we can refine the enforcement phase of an agent in such a way, that the assignments to the priority variables ensure that the enforcement properties of the vigilant agent hold. This process can be automated and effectively *compiles* a policy specification into the corresponding enforcement code, that guarantees the compliance of the system with its security policy. This is explained in the following.

9.4.1 Determining the Required Enforcement History

To be able to evaluate the rules, the history of the execution is needed. It is not efficient to store the history of the whole execution. To keep the history to a minimum, we transform the policy rules into a normal form, that allows us to compute the maximal history that is needed for each variable that is referenced in the rules.

Given the rule premise it is possible to determine the history of the referenced variables that is needed for the enforcement of the rule. For example the following premises of rules:

```
1 consequence when 0 : x = 1
2 consequence when 2 : x = 1
3 consequence when (1 : x = 1) ; (2: y = 5)
```

The first rule is a state-dependent rule, that states that `x` is *now* equal to 1. In this case no history for the variable `x` is required for the enforcement. The second rule is history dependent, and states that the value of `x` two time units in the past must have been equal

to 1 for the rule to fire. In this case the history of x must be kept for at least two previous states.

In the third rule two different variables, x and y, are referenced. The rule states that two time units in the past the variable y had the value 5 and in the state before x had the value 1. In this case the history for the variable y must be kept for at least the last 2 states and for the variable x for the last 3 states. To obtain the maximal time for which the history of a variable must be kept, we transform the rules into a normal form.

9.4.2 Rule Transformation

To compute the maximal history that must be kept for a rule to ensure that the rule can be enforced it is beneficial to transform the rule into disjunctive normal form. In the following we will use mathematical notation (e.g. $\wedge$) in preference of the ASCII representation (e.g. and) of the SANTA language, because of its compactness. The following rules can be applied:

$$f_1 \wedge (f_2 \vee f_3) \equiv (f_1 \wedge f_2) \vee (f_1 \wedge f_3) \tag{9.6}$$

$$f_1 \,;\, (f_2 \vee f_3) \equiv (f_1 \,;\, f_2) \vee (f_1 \,;\, f_3) \tag{9.7}$$

$$\textsf{sometime} f \equiv \textsf{true}; f \tag{9.8}$$

$$[t_1, \ldots, t_n] : f \equiv (t_1 : f) \vee \ldots \vee (t_n : f) \tag{9.9}$$

$$t : (f_1 \vee f_2) \equiv (t : f_1) \vee (t : f_2) \tag{9.10}$$

$$\textsf{exists}\ x < n : f \equiv (x = 0 \wedge f) \vee \ldots \vee (x = n \wedge f) \tag{9.11}$$

$$\textsf{forall}\ x < n : f \equiv (x = 0 \wedge f) \wedge \ldots \wedge (x = n \wedge f) \tag{9.12}$$

$$\textsf{if}\, w\, \textsf{then}\, f\, \textsf{else}\, g \equiv (w \wedge f) \vee (\neg w \wedge g) \tag{9.13}$$

We assume for (9.8) that the overall premise of the rule is finite. This is a reasonable assumption for the enforcement of rules, because policy decisions in infinite time are not of any practical interest. The syntax of rules allows only bounded quantification, viz. quantifications can be expanded into a finite conjunction or disjunction. For the static timing analysis we only consider the explicit and implicit time information in the premise. We do not consider the satisfiability of state-formulae and assume that any state-formula (except false) is satisfiable. For example the following premise

```
([1,3,7]:x=0) and ((2: (sometime y=2)) ; ([0,1]:z=0))
```

can be transformed to a disjunctive normal form by applying the above rules:

$$([1,3,7] : x = 0) \wedge ((2 : (\Diamond(y = 2))) \,;\, ([0,1] : z = 0))$$

Substitution of $\Diamond f$ with $\mathsf{true} \,;\, f$ (9.8) and expansion of the times (9.9):

$$((1 : x = 0) \vee (3 : x = 0) \vee (7 : x = 0)) \wedge ((2 : (\mathsf{true} \,;\, y = 2)) \,;\, ((0 : z = 0) \vee (1 : z = 0))$$

Applying (9.7) on the second conjunct:

$$\begin{aligned}&((1 : x = 0) \vee (3 : x = 0) \vee (7 : x = 0)) \wedge \\ &(((2 : (\mathsf{true} \,;\, y = 2)) \,;\, (0 : z = 0)) \vee ((2 : (\mathsf{true} \,;\, y = 2)) \,;\, (1 : z = 0)))\end{aligned}$$

Applying (9.6):

$$\begin{aligned}&((1 : x = 0) \wedge ((2 : (\mathsf{true} \,;\, y = 2)) \,;\, (0 : z = 0))) \vee \\ &((3 : x = 0) \wedge ((2 : (\mathsf{true} \,;\, y = 2)) \,;\, (0 : z = 0))) \vee \\ &((7 : x = 0) \wedge ((2 : (\mathsf{true} \,;\, y = 2)) \,;\, (0 : z = 0))) \vee \\ &((1 : x = 0) \wedge ((2 : (\mathsf{true} \,;\, y = 2)) \,;\, (1 : z = 0))) \vee \\ &((3 : x = 0) \wedge ((2 : (\mathsf{true} \,;\, y = 2)) \,;\, (1 : z = 0))) \vee \\ &((7 : x = 0) \wedge ((2 : (\mathsf{true} \,;\, y = 2)) \,;\, (1 : z = 0)))\end{aligned}$$

9.4.3 Determining the Chopping Points

To determine the length of the premise the following relations are used:

$$0 \leq \mathit{Length}(f) \leq \mathsf{T} \tag{9.14}$$

The maximum length of any interval defined by the premise of the rule is the enforcement time T.

$$\mathit{Length}(t : f) = t = \mathit{Length}(f) \tag{9.15}$$

$$\mathit{Length}(f ; g) = \mathit{Length}(f) + \mathit{Length}(g) \tag{9.16}$$

$$\mathit{Length}(f \wedge g) = \mathit{Length}(f) = \mathit{Length}(g) \tag{9.17}$$

We can use the relations between the lengths $Length(f)$ to determine the overall length of a premise or to show that a premise cannot be satisfied. For the above example we can establish that:

- The first disjunct is not satisfiable because $Length(1 : x = 0) = 1$ and $Length((2 : (\mathsf{true} \,;\, y = 2)) \,;\, (0 : z = 0))) = 2$. For the conjunction of both, the times must be equal (see (9.17)). The length of an interval cannot be 1 and 2 at the same time. Similarly the second, third, 4th and 6th disjunct are not satisfiable.

- The 5th disjunct is satisfiable. The timings that can be established are:

$$Length(3 : x = 0) = 3$$
$$Length((2 : (\mathsf{true} \,;\, y = 2)) \,;\, (1 : z = 0)) = 3$$
$$Length(2 : (\mathsf{true} \,;\, y = 2)) = 2 = (Length(\mathsf{true}) + Length(y = 0))$$
$$Length(1 : z = 0) = 1 = Length(z = 0)$$

In this case the timing $Length(\mathsf{true})$ and $Length(y = 0)$ cannot be established. However, we know that their sum is equal to 2. Together with (9.14) this leads to a finite number of combinations:

$$Length(\mathsf{true}) = 0 \text{ and } Length(y = 0) = 2$$
$$Length(\mathsf{true}) = 1 \text{ and } Length(y = 0) = 1$$
$$Length(\mathsf{true}) = 2 \text{ and } Length(y = 0) = 0$$

If the chopping point is not determined by either the left-hand side or the right-hand side, then it can be at any state in the interval. Since the overall length of the interval is always less than T this case can be addressed by making the finite number of possible combinations explicit:

$$(2 : (\mathsf{true} \,;\, y = 2))$$

is transformed by explicitly specifying the possible chop points to:

$$(2 : (((0 : \mathsf{true}) \,;\, (2 : y = 2)) \vee ((1 : \mathsf{true}) \,;\, (1 : y = 2)) \vee ((2 : \mathsf{true}) \,;\, (0 : y = 2))))$$

The new premise is again brought into disjunctive normal form:

$$\begin{aligned}(3 : x = 0) \wedge ((2 : (((0 : \mathsf{true}) \,;\, (2 : y = 2)) \vee \\ ((1 : \mathsf{true}) \,;\, (1 : y = 2)) \vee \\ ((2 : \mathsf{true}) \,;\, (0 : y = 2)))) \,;\, (1 : z = 0))\end{aligned}$$

Applying (9.10) and (9.7)

$$\begin{aligned}(3 : x = 0) \wedge ((2 : ((0 : \mathsf{true}) \,;\, (2 : y = 2)) \,;\, (1 : z = 0)) \vee \\ (2 : ((1 : \mathsf{true}) \,;\, (1 : y = 2)) \,;\, (1 : z = 0)) \vee \\ (2 : ((2 : \mathsf{true}) \,;\, (0 : y = 2)) \,;\, (1 : z = 0)))\end{aligned}$$

Applying (9.6)

$$\begin{aligned}((3 : x = 0) \wedge (2 : ((0 : \mathsf{true}) \,;\, (2 : y = 2)) \,;\, (1 : z = 0))) \vee \\ ((3 : x = 0) \wedge (2 : ((1 : \mathsf{true}) \,;\, (1 : y = 2)) \,;\, (1 : z = 0))) \vee \\ ((3 : x = 0) \wedge (2 : ((2 : \mathsf{true}) \,;\, (0 : y = 2)) \,;\, (1 : z = 0)))\end{aligned}$$

This ensures, that all chopping points in the premise are deterministic. Since the premise describes a past behaviour from the view-point of the state in which the decision is made, we take this state as the fix-point when discussing history variables. We label the final state of the interval with σ_0^* and the state in which the enforcement started with σ_T^*. The meaning of σ_i^* is then the state that is i in the past of the state for which the policy decision is made.

It is now easy to compute the state of each chopping point, by starting at the rightmost element of the sequence, working towards the leftmost element adding each time the length of the sequence element. For the example we annotated the chopping points with the state below:

$$\begin{aligned}((3 : x = 0) \wedge (2 : ((0 : \mathsf{true}) \,;^{\sigma_3^*}\, (2 : y = 2)) \,;^{\sigma_1^*}\, (1 : z = 0))) \vee \\ ((3 : x = 0) \wedge (2 : ((1 : \mathsf{true}) \,;^{\sigma_2^*}\, (1 : y = 2)) \,;^{\sigma_1^*}\, (1 : z = 0))) \vee \\ ((3 : x = 0) \wedge (2 : ((2 : \mathsf{true}) \,;^{\sigma_1^*}\, (0 : y = 2)) \,;^{\sigma_1^*}\, (1 : z = 0)))\end{aligned}$$

With this information at hand it is straight forward to determine the maximum history that must be kept for each variable to be able to evaluate the premise of the rule: It is

the earliest chopping point left to the state formula in which the variable occurs. For the example the maximal history for x is 3; the maximal history for y is 3; and the maximal history of z is 1.

9.4.4 Transformation of the Vigilant Agent Example

The transformation of the rules is not detailed here, because it follows the same lines as the previously discussed premise. Only the obligation rule is of interest, as it references the enforcement time T.

```
oblige (ag,ag,inc) when [T..2,2]: sometime (x<3)
```

In the premise of this rule, we expand the **sometime** to:

$$[\mathsf{T}..2,2] : \mathsf{true} \,;\, (x,3)$$

T is the current enforcement time, viz. the abstract time since the enforcement of the rule started in the state where the consequence of the rule is determined. In this case the explicit timing of the rule states that any interval of length T to at most 2 in which sometimes x is less than 3. Knowing that T in the first state of the enforcement is 0 we can expand this to:

$$\begin{aligned}
&(\mathsf{T} = 0 \wedge [0..2] : (\mathsf{true} \,;\, (x < 3))) \vee \\
&(\mathsf{T} = 1 \wedge [1..2] : (\mathsf{true} \,;\, (x < 3))) \vee \\
&(\mathsf{T} = 2 \wedge [2..2] : (\mathsf{true} \,;\, (x < 3))) \vee \\
&(2 : (\mathsf{true} \,;\, (x < 3)))
\end{aligned}$$

When expanding the times of the first disjunct, viz. [0..2] : to:

$$(\mathsf{T} = 0 \wedge (0 : (\mathsf{true} \,;\, (x < 3)) \vee 1 : (\mathsf{true} \,;\, (x < 3)) \vee 2 : (\mathsf{true} \,;\, (x < 3))))$$

it is clear that the disjunct $1 : \ldots$ and $2 : \ldots$ cannot be satisfied, because every length must be less than or equal to T (see (9.14)). Since T is equal to 0, they cannot be satisfied. Consequently the premise can be reduced to:

$$\begin{aligned}
&(\mathsf{T} = 0 \wedge 0 : (\mathsf{true} \,;\, (x < 3))) \vee \\
&(\mathsf{T} = 1 \wedge 1 : (\mathsf{true} \,;\, (x < 3))) \vee \\
&(2 : (\mathsf{true} \,;\, (x < 3))) \vee
\end{aligned}$$

The disjunct $(\mathsf{T} = 2 \wedge \ldots$ is implies the last disjunct $(2 : (\mathit{true}\ldots$ and can therefore be removed. Again the chopping point of the sequence is not yet deterministic and must be made explicit as discussed previously.

$$
\begin{aligned}
&(\mathsf{T} = 0 \wedge 0 : (x < 3)) \vee \\
&(\mathsf{T} = 1 \wedge 1 : (x < 3)) \vee \\
&(\mathsf{T} = 1 \wedge (1 : \mathsf{true})\,;(0 : (x < 3))) \vee \\
&(2 : (x < 3)) \vee \\
&((1 : \mathsf{true})\,;(1 : (x < 3))) \vee \\
&((2 : \mathsf{true})\,;(0 : (x < 3)))
\end{aligned}
$$

The result of the transformation is that the history of the agent variable x must be kept for 2 past states. This is needed for the enforcement of the obligation rule.

9.4.5 Refining the Enforcement Time

The enforcement phase as described in Section 5.3.4 guarantees that all agent variables are maintained and that the phase takes only finite time to execute. The abstract specification of the enforcement time T is implemented in the enforcement phase. In the semantics of policies the enforcement time is modelled as a static variable that has the same value as *Clock* in the final state of the interval. The specification of a simple policy states that *Clock* is incremented by one from each policy state to the next. We established earlier in Chapter 8, that the enforcement phase is executed once between each state in the policy enforced by a vigilant agent. We can therefore model *Clock* as an enforcement auxiliary variable that is initialised to -1 and incremented in the beginning of the enforcement phase. Figure 9.4 depicts the relation between the implementation level and the abstraction level of the policy.

Formally we initialise the variable *Clock* in the initialisation phase to the value -1 and refine the enforcement phase enf_a to:

$$
\begin{aligned}
&enf_{ag} \sqsubseteq \\
&enf^0_{ag} = [\![Clock := Clock + 1]\!]_{V_{ag}}\,;\, enf_{ag}
\end{aligned}
$$

To maintain the variable *Clock* during the rest of the agent execution we add it to the set of enforcement auxiliary variables $V_{ag,enf}$. The proof that this is a refinement is along the same lines as for the finite prefix introduction for a temporal assignment (see Theorem 9 on page 224).

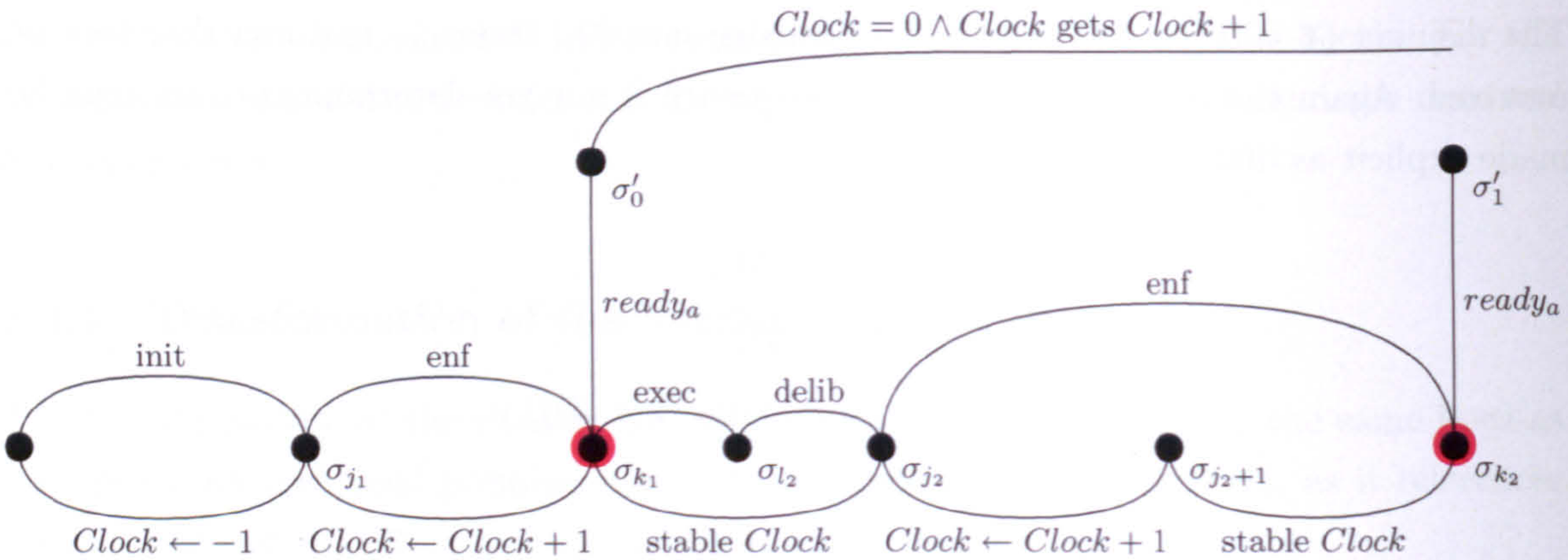


Figure 9.4: Enforcement Time

9.4.6 Maintaining the History

To maintain the history of the variables, we introduce for each variable that is used in the premise of a rule a list that holds its history. We denote this list in the following by H_v, where v is the identifier of the variable. As a convention we keep in the first element the current value of the variable (viz. $H_v[0] = v$).

The refinement of the enforcement phase to maintain the history of the variables is similar to the refinement for the enforcement time. For each variable that is used in the policy, we introduce an enforcement auxiliary variable that holds its history. The history is maintained by shifting the values in the list once in each execution of the enforcement phase. The associated assignments are all finite and can therefore be introduced in the same way as the enforcement time.

If the maximal required history of a variable is a constant, the list is of a fixed size. If the maximal required history is a variable, then the size of the list grows linearly and is at most of size T. For the vigilant agent example we established that the maximal required history of the agent variable x is 2. We can therefore refine the agents enforcement phase to:

$$
\begin{aligned}
& enf^0_{ag} \sqsubseteq \\
& enf^1_{ag} = [\![T := T + 1]\!]_{V_{ag}}; \\
& \qquad [\![H_x[2], H_x[1], H_x[0] \leftarrow H_x[1], H_x[0], x]\!]_{V_{ag}}; \\
& \qquad enf_{ag}
\end{aligned}
$$

Consequently the list $H_x[i]$ denotes the value that x had i time units in the past, provided that $i \leq T$. To keep the values of the history variables stable we add them to the set of enforcement auxiliary variables $V_{ag,enf}$.

9.4.7 Refining Obligation Rules

If a vigilant agent has an obligation we must ensure that the enforcement property $EP_{ag,oblig}$ holds (see Section 8.5.4). The enforcement property states that the agent can only successfully execute an action, if it is either obliged to do so or if it cannot possibly execute another action that it is obliged to execute.

This is ensured by assigning the value -1 to the priority variable corresponding to the obliged action. The negative value ensures, that the action has a higher priority than any that has been assigned in the deliberation phase of the agent. The functional guards of the agent's actions (see (5.13), page 118) ensure that — provided the precondition of the obliged action holds — no other action but an obliged action can be enabled. If the vigilant agent has more than one obligation, the agent will choose to comply with one of them.

To reassign the priority of the obliged actions, we refine the enforcement phase:

$$
\begin{aligned}
&enf^1_{ag} \sqsubseteq \\
&enf^2_{ag} = [\![Clock := Clock + 1]\!]_{V_{ag}}; \\
&[\![H_x[2], H_x[1], H_x[0] \leftarrow H_x[1], H_x[0], x]\!]_{V_{ag}}; \\
&[\![\textsf{if } oblig(ag, ag, inc) \textsf{ then } \Pi_{ag,inc} := -1]\!]_{V_{ag}}; \\
&enf_{ag}
\end{aligned}
$$

Here the condition for the obligation is expressed by $oblig(ag, ag, inc)$ and is determined by the premise of the rule. The premise is already in normal form and can be transformed into an expression on the history variables using the following rules:

$$(t_1 : f_1)\,;(t_2 : f_2) \equiv ((t_1 : f_1)\,;(t_2 : \textsf{true})) \wedge ((t_1 : \textsf{true})\,;(t_2 : f_2)) \tag{9.18}$$

This is possible because the chopping point is deterministic. For the second line we can omit the length of the prefix for the second conjunct, because the overall length is determined by the first conjunct. For the special case of a state-formula:

$$(t_1 : w_1)\,;(t_2 : \textsf{true}) \equiv ((t_1 + t_2) : w_1) \tag{9.19}$$

This is because a state-formula holds over an interval, if it holds in the initial state of that interval. These two equations allow us to transform all sequences in the premise of a rule into conjunctions where each conjunct is of the form:

$$(t_1 : \mathsf{true})\ ;\ (t_2 : w)$$

This means that there is a suffix of length t_2 for which w holds in the initial state. In the rule syntax a state formula is limited to be a boolean expression, viz. the evaluation of the expression in the state $\sigma^*_{t_2}$ is the same as the evaluation of the expression on the values of the variables in state $\sigma^*_{t_2}$. These values are stored in the history:

$$(t_1 : \mathsf{true})\ ;\ (t_2 : w) = 0 : (\mathsf{T} \geq (t_1 + t_2) \wedge w[H_{v_i}[t_2]/v_i])$$

where v_i are the free variables in w. The additional constraint that T is greater than or equal to $t_1 + t_2$ captures the length requirement. In the state-formula w all variables are substituted with the history variable for the time t_2. This state formula evaluated in the last state of the interval is *true* iff the same expression evaluated in state $\sigma^*_{t_2}$ was *true*.

For the obligation rule we apply the above transformation as follows:

$$\begin{aligned}
&(\mathsf{T} = 0 \wedge 0 : (\mathsf{T} \geq 0 \wedge H_x[0] < 3)) \vee (\mathsf{T} = 1 \wedge 0 : (\mathsf{T} \geq 1 \wedge H_x[1] < 3)) \vee \\
&(\mathsf{T} = 1 \wedge 0 : (\mathsf{T} \geq 1 \wedge H_x[0] < 3)) \vee (0 : (\mathsf{T} \geq 2 \wedge H_x[2] < 3)) \vee \\
&(0 : (\mathsf{T} \geq 2 \wedge H_x[1] < 3)) \vee (0 : (\mathsf{T} \geq 2 \wedge H_x[0] < 3))
\end{aligned}$$

It is important to note that the variable T references a local static variable that has the same value as the variable *Clock* in the final state of the interval. Consequently the above premise is equivalent to:

$$\begin{aligned}
&(Clock = 0 \wedge H_x[0] < 3) \vee (Clock = 1 \wedge H_x[1] < 3) \vee \\
&(Clock = 1 \wedge H_x[0] < 3) \vee (Clock \geq 2 \wedge H_x[2] < 3) \vee \\
&(Clock \geq 2 \wedge H_x[1] < 3) \vee (Clock \geq 2 \wedge H_x[0] < 3)
\end{aligned}$$

interpreted in the same state in which the consequence of the rule is determined. The above is the condition that is equivalent[1] with $oblig(ag, ag, inc)$. The enforcement phase is then:

[1]Provided that the policy is complete as discussed in Section 8.4.

$$
\begin{aligned}
enf^2_{ag} = & [Clock := Clock + 1]]_{V_{ag}}; \\
& [H_x[2], H_x[1], H_x[0] \leftarrow H_x[1], H_x[0], x]]_{V_{ag}}; \\
& [\text{if } (Clock = 0 \wedge H_x[0] < 3) \vee (Clock = 1 \wedge H_x[1] < 3) \vee \\
& \quad (Clock = 1 \wedge H_x[0] < 3) \vee (Clock \geq 2 \wedge H_x[2] < 3) \vee \\
& \quad (Clock \geq 2 \wedge H_x[1] < 3) \vee (Clock \geq 2 \wedge H_x[0] < 3) \text{ then } \Pi_{ag,inc} := -1]]_{V_{ag}}; \\
& enf_{ag}
\end{aligned}
$$

This enforces the obligation for the agent `ag` to execute the action `inc` as stated in the vigilant agent example.

9.4.8 Refining Authorisations

Authorisations are refined *after* obligations. The intuition is that a denial is stronger than an obligation. By refining authorisation rules after the obligation, the assignment made by the obligation can be overruled. This captures the specification of the enforcement property for obligations (see (8.8) on page 208). The mechanism is similar to the one for obligations.

$$
\begin{aligned}
& enf^2_{ag} \sqsubseteq \\
& enf^3_{ag} = [Clock := Clock + 1]]_{V_{ag}}; \\
& \quad [H_x[2], H_x[1], H_x[0] \leftarrow H_x[1], H_x[0], x]]_{V_{ag}}; \\
& \quad [\text{if } (Clock = 0 \wedge H_x[0] < 3) \vee (Clock = 1 \wedge H_x[1] < 3) \vee \\
& \qquad (Clock = 1 \wedge H_x[0] < 3) \vee (Clock \geq 2 \wedge H_x[2] < 3) \vee \\
& \qquad (Clock \geq 2 \wedge H_x[1] < 3) \vee (Clock \geq 2 \wedge H_x[0] < 3) \text{ then } \Pi_{ag,inc} := -1]]_{V_{ag}}; \\
& \quad [\text{if } \neg autho(ag, ag, inc) \text{ then } \Pi_{ag,inc} := 0]]_{V_{ag}}; \\
& \quad [\text{if } \neg autho(ag, ag, dbl) \text{ then } \Pi_{ag,dbl} := 0]]_{V_{ag}}; \\
& \quad enf_{ag}
\end{aligned}
$$

We know that the conflict resolution rule in the policy gives precedence to denials. Since the rule is state-dependent we can also write:

$$
\begin{aligned}
enf^3_{ag} = & [\![Clock := Clock + 1]\!]_{V_{ag}}; \\
& [\![H_x[2], H_x[1], H_x[0] \leftarrow H_x[1], H_x[0], x]\!]_{V_{ag}}; \\
& [\![\textsf{if}\,(Clock = 0 \wedge H_x[0] < 3) \vee (Clock = 1 \wedge H_x[1] < 3) \vee \\
& \quad (Clock = 1 \wedge H_x[0] < 3) \vee (Clock \geq 2 \wedge H_x[2] < 3) \vee \\
& \quad (Clock \geq 2 \wedge H_x[1] < 3) \vee (Clock \geq 2 \wedge H_x[0] < 3)\ \textsf{then}\ \Pi_{ag,inc} := -1]\!]_{V_{ag}}; \\
& [\![\textsf{if}\,\neg(autho^+(ag, ag, inc) \wedge \neg autho^-(ag, ag, inc))\ \textsf{then}\ \Pi_{ag,inc} := 0]\!]_{V_{ag}}; \\
& [\![\textsf{if}\,\neg(autho^+(ag, ag, dbl) \wedge \neg autho^-(ag, ag, dbl))\ \textsf{then}\ \Pi_{ag,dbl} := 0]\!]_{V_{ag}}; \\
& enf_{ag}
\end{aligned}
$$

Since $autho^+(ag, ag, inc)$ is always *true* and $autho^-(ag, ag, inc)$ is always *false* we can remove the if construct ((if false then f else empty) $\equiv$ empty *and* (empty ; f) $\equiv$ f). We also know that $autho^+(ag, ag, dbl)$ is always *true*. Consequently:

$$
\begin{aligned}
enf^3_{ag} = & [\![Clock := Clock + 1]\!]_{V_{ag}}; \\
& [\![H_x[2], H_x[1], H_x[0] \leftarrow H_x[1], H_x[0], x]\!]_{V_{ag}}; \\
& [\![\textsf{if}\,(Clock = 0 \wedge H_x[0] < 3) \vee (Clock = 1 \wedge H_x[1] < 3) \vee \\
& \quad (Clock = 1 \wedge H_x[0] < 3) \vee (Clock \geq 2 \wedge H_x[2] < 3) \vee \\
& \quad (Clock \geq 2 \wedge H_x[1] < 3) \vee (Clock \geq 2 \wedge H_x[0] < 3)\ \textsf{then}\ \Pi_{ag,inc} := -1]\!]_{V_{ag}}; \\
& [\![\textsf{if}\ autho^-(ag, ag, dbl))\ \textsf{then}\ \Pi_{ag,dbl} := 0]\!]_{V_{ag}}; \\
& enf_{ag}
\end{aligned}
$$

We convert the premise of the negative authorisation rule analogously to the obligation rule. $autho^-(ag, ag, dbl)$ is then replace by the expression on history variables:

$$
\begin{aligned}
enf^3_{ag} =& [\![Clock := Clock + 1]\!]_{V_{ag}};\\
& [\![H_x[2], H_x[1], H_x[0] \leftarrow H_x[1], H_x[0], x]\!]_{V_{ag}};\\
& [\![\textbf{if}\,(Clock = 0 \wedge H_x[0] < 3) \vee (Clock = 1 \wedge H_x[1] < 3) \vee\\
& \quad (Clock = 1 \wedge H_x[0] < 3) \vee (Clock \geq 2 \wedge H_x[2] < 3) \vee\\
& \quad (Clock \geq 2 \wedge H_x[1] < 3) \vee (Clock \geq 2 \wedge H_x[0] < 3)\ \textbf{then}\ \Pi_{ag,inc} := -1]\!]_{V_{ag}};\\
& [\![\textbf{if}\ Clock \geq 1 \wedge H_x[1] < 5\ \textbf{then}\ \Pi_{ag,dbl} := 0]\!]_{V_{ag}};\\
& enf_{ag}
\end{aligned}
$$

9.4.9 Refining Integrity Rules

The enforcement of integrity rules cannot be implemented in the enforcement phase of the agent, because it requires the results of the computation. This is available after the execution of the action statement on the local variables (see Section 5.3.5). The semantics of a local action defines that the action does either succeed or fail (see Equation (5.16)) where both the success and the failure are defined as a temporal multiple assignment. The definitions are included again here for the readers convenience

$$stat_{a,x} \mathrel{\widehat{=}} \exists v'_0, \ldots, v'_j \cdot ([\![\ \boxed{v'_0, \ldots v'_j \leftarrow\ v_0, \ldots, v_j}\ ]\!]_{V_a^+}\ ;\ [\![S'_x]\!]_{V_a^+}\ ;\ (succeed_{a,x} \oplus fail_{a,x}))$$

$$succeed_{a,x} \mathrel{\widehat{=}} [\![\ \boxed{done_{a,x}, \overline{done_{a,x}}, v_0, \ldots, v_j \leftarrow\ \texttt{true}, \overline{false}, v'_0, \ldots, v'_j}\ ]\!]_{V_a^+}$$

$$fail_{a,x} \mathrel{\widehat{=}} [\![\ \boxed{failed_{a,x}, \overline{failed_{a,x}} \leftarrow\ \texttt{true}, \overline{false}}\ ]\!]_{V_a^+}$$

To enforce the integrity rule for an agent's action we refine the nondeterministic choice $(succeed_{a,x} \oplus fail_{a,x})$ by a deterministic choice:

$$
\begin{aligned}
& stat_{a,x} \sqsubseteq\\
& stat^0_{a,x} = \exists v'_0, \ldots, v'_j \cdot ([\![\ \boxed{v'_0, \ldots v'_j \leftarrow\ v_0, \ldots, v_j}\ ]\!]_{V_a^+}\ ;\ [\![S'_x]\!]_{V_a^+};\\
& \qquad \textbf{if}\ integ(a, a, x)\ \textbf{then}\ succeed_{a,x}\ \textbf{else}\ fail_{a,x})
\end{aligned}
$$

The primed variables in the policy rule reference the local variables, that store the temporary result of the execution. So for example for the enforcement of the integrity rule

of the vigilant agent example the semantics of the action `inc` would be:

$$stat_{ag,inc} = \exists x' \cdot ([\![\boxed{x' \leftarrow x}]\!]_{V_{ag}^+} ; [\![x' := x' + 1]\!]_{V_{ag}^+} ; \\ \textbf{if } x' < 20 \textbf{ then } succeed_{ag,inc} \textbf{ else } fail_{ag,inc})$$

If the premise of the rule does reference the history of the execution, we transform the premise into an expression on history variables as shown for the obligation and authorisation.

9.4.10 Automation

The process of establishing the maximal time for which the history of variables must be kept to make a decision, as well as the transformation into the normal form can be automated. The aim of the automation is to derive the enforcement code for a simple policy automatically. This means that the policy is compiled into the correct enforcement code, rather than interpreted at runtime. This has the benefit that guarantees can be given on the time that is needed to make the policy decision.

The formal grounding of the tool allows for the optimisation of the rules and some basic satisfiability checks. Figure 9.5 shows an example of the prototype of the tool.

```
1  Available actions:
2  ?      print menu      prints the menu descriptions of all available
       actions
3  times  toggle times    Toggles the use of time intervals [ON][OFF]
4  r      enter rule      prompts to enter a rule
5  p      dump tree       Prints the parse tree on the screen.
6  1      sub sometime    substitutes sometimes f with true ; f
7  e1     Expand ANDOR    (f or g) and h  --> (f and h) or (f and g)
8  e2     Expand CHOPOR   (f or g) ; h  --> (f ; h) or (g ; h)
9  t      Annotate times  computes the timings in the rule
10 d      Disj. Normal    transforms f into disjunctive normal form.
11 f      format tree     formats the current parse tree
12 l      label nodes     Relabels the nodes in the tree.
13 q      quit program    quits the program
14
15 (choose option ? for help) >r
16 Reading a rule from standard input (terminate with '.' and press enter)
       ...
17 [10]:x=0 and ([1]:y=0 ; [0..8]:z=0).
18
19 (choose option ? for help) >t
20 This rule is not satisfiable
```

Figure 9.5: Example of Automated Rule Analysis

The current version of the tool, does not support all transformation rules that are needed to obtain the disjunctive normal form. For example the explicit timings are not transformed into a disjunction. The tool by default computes the length of individual sub-intervals on an minimum/ maximum basis. The example in Figure 9.5 defines a premise of length 10, where $x = 0$ in the initial state and a sequence where $y = 0$ in the prefix of length 1 and $z = 0$ in the suffix of any length between 0 to 8. The analysis of the minimal and maximal length shows that the sequence must be between length 1 and length 9. In conjunction with an interval of length 10 this is not satisfiable.

With explicit timings provided the tool can also compute possible combinations of timings that result from sequences. This allows to optimise the premise to omit cases that are known to be not satisfiable. Figure 9.6 provides a more complex example:

```
1  Reading a rule from standard input (terminate with '.' and press enter)
       ...
2  [5,7,11,13,17,23]:([1,2,0,5]:x=0 ; [2,5,13]:y=0).
3
4  (choose option ? for help) >t
5  This rule is not satisfiable on intervals shorter than 5
6  This rule is not satisfiable on intervals longer than 13
7  This rule is only satisfiable on intervals of length t in [5, 7, 13]
8
9  (choose option ? for help) >p+
10   0: Start  [min= 5; max= 13; times= [5, 7, 13]]
11   0.0: TimedNode  [min= 5; max= 13; times= [5, 7, 13]]
12   0.0.0: ListNode
13   0.0.0.0: IntConstNode = 5
14   0.0.0.1: IntConstNode = 7
15   0.0.0.2: IntConstNode = 11
16   0.0.0.3: IntConstNode = 13
17   0.0.0.4: IntConstNode = 17
18   0.0.0.5: IntConstNode = 23
19   0.0.1: ChopNode  [min= 5; max= 13; times= [5, 7, 13]]
20   0.0.1.0: TimedNode  [min= 0; max= 5; times= [0, 2, 5]]
21   0.0.1.0.0: ListNode
22   0.0.1.0.0.0: IntConstNode = 1
23   0.0.1.0.0.1: IntConstNode = 2
24   0.0.1.0.0.2: IntConstNode = 0
25   0.0.1.0.0.3: IntConstNode = 5
26   0.0.1.0.1: EQNode  [min= 0; max= 5; times= [0, 2, 5]]
27   0.0.1.0.1.0: Id name= x; val=0
28   0.0.1.0.1.1: IntConstNode = 0
29   0.0.1.1: TimedNode  [min= 2; max= 13; times= [2, 5, 13]]
30   0.0.1.1.0: ListNode
31   0.0.1.1.0.0: IntConstNode = 2
32   0.0.1.1.0.1: IntConstNode = 5
33   0.0.1.1.0.2: IntConstNode = 13
34   0.0.1.1.1: EQNode  [min= 2; max= 13; times= [2, 5, 13]]
35   0.0.1.1.1.0: Id name= y; val=0
36   0.0.1.1.1.1: IntConstNode = 0
```

Figure 9.6: Example of Automated Rule Analysis

In this example the tool discovers that only intervals of length 5, 7 or 13 can possibly satisfy the premise. More interestingly is that it also concludes that the prefix of the sequence cannot be of length 1 as there is no combination with any potential length of the suffix that could result in any of the explicitly defined overall length. This can be seen in

the annotated parse-tree (line 20) where the list of potential times does not contain 1.

Beside its application for the automatic generation of enforcement code, the tool can also be integrated in the policy specification process to provide an early feedback on the satisfiability of rules.

9.4.11 Enforcing Composed Policies

Policies can be composed along the temporal and structural axis. The enforcement along the structural axis is solved similarly as shown for the enforcement of hybrid access control policies (positive and negative authorisation combined by a decision rule). Whilst the enforcement may require additional variables, it does not increase the complexity of the approach.

For the composition along the temporal axis, the transition between two simple policies is dependent on time or the occurrence of events. When refining a complex policy into enforcement code the event is checked in every execution of the enforcement phase. If the event occurs, the enforcement timer is reset and the policy changes to the next policy in the sequence. For example:

```
policy p  : /* */ end
policy q  : /* */ end
policy pq : (unless x >= 10: p) ; q end
```

In this case the event for the change is $x \geq 10$. In the enforcement phase the change from policy p to policy q takes place when the event is observed. To capture the currently enforced policy we introduce an auxiliary variable *seq* that denotes the currently enforced policy. Initially p is enforced, viz. *seq* is initialised to p. The enforcement phase is then refined into a check on the event:

$$
\begin{aligned}
enf_a^3 = & [\![Clock := Clock + 1]\!]_{V_a}; \\
& [\![\textbf{if}\,(seq = p \wedge x \geq 10)\ \textbf{then}\ seq, Clock \leftarrow q, 0]\!]_{V_a}; \\
& [\![\textbf{if}\ (seq = p)\ \textbf{then}\ \text{refinement of}\ p\ \textbf{else} \\
& \ \textbf{if}\ (seq = q)\ \textbf{then}\ \text{refinement of}\ q]\!]_{V_a}
\end{aligned}
$$

The sequential composition of policies is implemented as a finite state machine where *seq* denotes the current state (viz. the currently enforced policy). With every execution of the enforcement phase it is checked whether a transition is made, or if the enforcement remains in its current state. By refinement of p we mean the update of the history variables required for the enforcement of p and the conditional assignments of the priority variables

that reflect the policy decisions as it was discussed in the previous subsections.

This is an example of compositional refinement, where first the temporal policy composition is refined and then the simple policies that are components of the composition. The refinement of the composition is essentially an automata and has similarities with *security automata* [122], however a state does not denote the acceptable system states, but the current state of the enforced policy.

9.5 Summary

In this chapter we have introduced refinement as implication. A refined specification implies its original specification. We presented some refinement rules for which this relation is not obvious and proved the rules to be correct. In Section 9.3 the rules have been applied to refine the $SMAS_1$ example to a concrete and implementable program. The refinement of an $SMAS_2$ was not detailed in this chapter, as it follows the same lines as the $SMAS_1$ example.

We then focussed on the refinement of a vigilant agent. The vigilant agent enforcement mechanism defines the effect that a policy has on the agent's execution in form of enforcement properties. We have shown how a policy can be brought into a disjunctive normal form and how the maximal history for all variables referenced in the policy can be obtained. We further detailed, how the policy can be transformed into an expression on history variables.

Refining every policy rule into concrete enforcement code manually would be a significant overhead to the development process. Some of the refinement rules can be extended to be used at the syntactic level. This would reduce the overhead. Ultimately we envision the automation of the policy refinement so that policies can easily be enforced by a dedicated software component in the system. Although we did not explicitly study the scalability of the approach we believe it will scale well with the size of policies and the execution time. The approach broadly separated the maintenance of history variables and the evaluation of simple boolean conditions to determine the policy decision. The former optimises the amount of administration and the memory usage that is associated with the enforcement of history-based access control, the latter can be optimised using standard techniques, e.g. found in logic programming or compiler theory for the optimisation of boolean expressions.

The history variables have been introduced as part of the agent's state variables and the enforcement phase has been refined to maintain the history. We have shown how obligation and authorisation rules are implemented as conditional assignments that depend on the expression of history variables. The enforcement of integrity rules differs in its

implementation from obligation and authorisation in that it refines the non-deterministic choice between an action failure or success with a conditional choice. The choice depends on the expression on history variables that was derived from the rule premise.

We then outlined how policies that are composed along the structural and temporal axis are enforced. Notably the composition along the temporal axis provides a performance benefit over more traditional approaches, because the set of rules that applies at a time is smaller, i.e. less rules must be checked. The process of transforming the policy rules into the corresponding enforcement code can be automated. We presented an initial prototype of the envisioned tool-support. The benefit of the automation is that the enforcement code can be pre-compiled and optimised before the policy is deployed. This will increase the performance of the enforcement and also allows to reason about the time that is needed to make policy decisions. The underlying semantics of the enforcement code can be used to prove that a policy decision can made under hard real-time constraints.

Chapter 10

Analysis

The objective of this chapter is to present tool-support for the analysis of policies. The tool is aimed to allow for the early and frequent validation of policies early in the development life-cycle. The aim is to test whether a policy captures the original intent of the policy designer.

10.1 Introduction

In this chapter we introduce the Security Policy Analysis Tool (SPAT) that allows for the early and frequent validation of policies during the development-life cycle. The analysis made with SPAT is based on a simulation of policies using the Tempura language [66, 99]. Tempura is an executable subset of ITL — this ensures that the semantics of the policies can be accurately captured. SPAT simulates a specific scenario of a system run by defining the observable events and the sequence and timing of their occurrences. Given this fixed scenario the policy is simulated and all policy decisions during the run are recorded for analysis.

The record of all policy decisions is kept by a Java [63, 93] program that provides a more convenient and extensible platform for the analysis of the results. SPAT itself is designed to support the development of a policy specification and its continuous validation within the same interface. This allows for a fast feedback to the policy designer and supports an iterative development approach for policies.

SPAT currently is still under development and is only available as a prototype. Due to resource limitations a full implementation that is up to industrial standards was not feasible. This means that the different modules in the SPAT architecture are not linked and the (straight-forward) translation between policy and the corresponding Tempura program must be performed manually.

The structure of the Chapter is as follows. In Section 10.2 we present the architecture of the SPAT tool together with an overview of the available features. In Section 10.3 we provide a case study that was published in [75]. We then show in Section 10.4 how SPAT can provide a graphical user interface for the composition of policies and in Section 10.5 the support for the analysis of Policy decisions. We conclude the chapter with a short summary in Section 10.6.

10.2 SPAT Architecture

SPAT is conceived to support the specification of a test scenario and a policy specification that is to be analysed for the scenario. Given this specification SPAT will simulate the scenario and visualise the effect of the policy. The policy decisions can be explained to the user in terms of policy rules that influenced the decision. An overview of the SPAT architecture is provided in Figure 10.1.

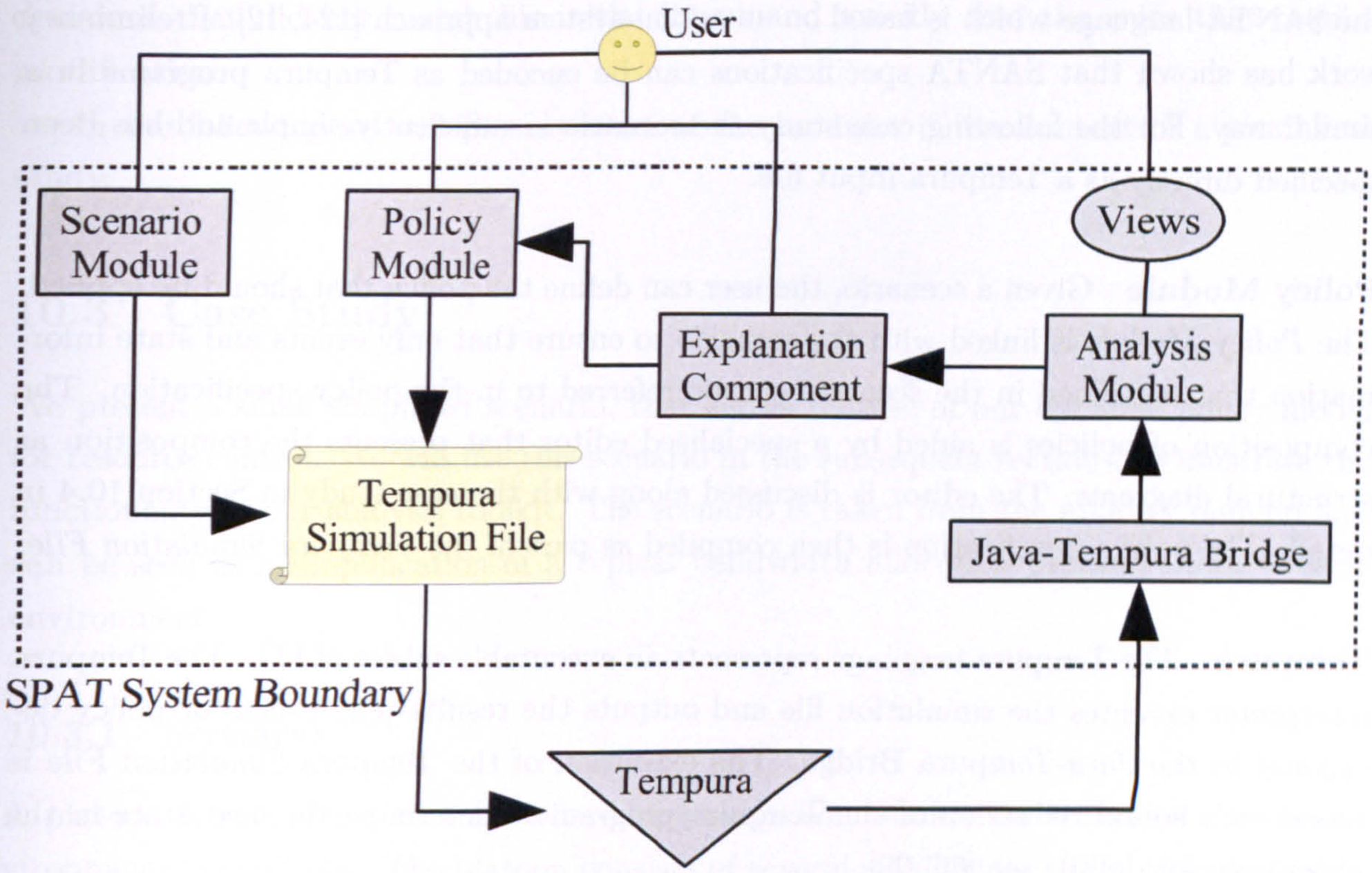


Figure 10.1: SPAT Architecture

SPAT consists of several modules that are responsible for different tasks in the policy specification process. SPAT is intended to be used in the development process of policy specification or for later policy revision. Typically when a policy is being developed there is already a high-level description of the system structure available. For example subjects, objects and actions are identified, however their concrete implementation is still unknown. Similarly for the later revision of a policy. A policy revision will occur typically during the development process or even after the system has been deployed. In this case the structure of the system is already in place and the policy designer has a concrete scenario at hand for which the original policy has proven to be insufficient.

Scenario Module The SPAT user first defines the basic structure of the system in form of a scenario. The scenario description contains the subjects, objects and actions in the system as well as events that are relevant to the policy. The scenario can be at a high level of abstraction by defining only the events and their timings or more concrete in form of a concrete SANTA program. The scenario is then compiled as part of the *Tempura Simulation File*.

Currently the *Scenario Module* allows only for the specification using a predecessor of

the SANTA-language which is based on an action-system approach [124, 12]. Preliminary work has shown that SANTA specifications can be encoded as Tempura programs in a similar way. For the following case study the scenario is sufficiently simple and has been specified directly as a Tempura input file.

Policy Module Given a scenario, the user can define the policy that should be applied. The *Policy Module* is linked with the scenario to ensure that only events and state information that is defined in the scenario can be referred to in the policy specification. The composition of policies is aided by a specialised editor that presents the composition as structural diagrams. The editor is discussed along with the case study in Section 10.4 in detail. The policy specification is then compiled as part of the *Tempura Simulation File.*

Tempura The Tempura language represents an executable subset of ITL. The Tempura interpreter executes the simulation file and outputs the results (e.g. events or policy decisions) to the *Java-Tempura Bridge.* The execution of the *Tempura Simulation File* is based on a sound reduction of the Tempura program to determine the next state in the execution; for details see [66, 99].

Java-Tempura Bridge The *Java-Tempura Bridge* is the communication interface between the *Analysis Module* written in Java and the Tempura Interpreter (implemented in *C*). It provides functions to access Java objects from the *Tempura Simulation File*, enabling event-recognition and visualisation.

Analysis Module The *Analysis Module* records the events and policy decisions during the simulation. It provides a basic interface for standard and proprietary plug-ins. The standard plug-ins in SPAT visualise the policy decisions in form of an access control matrix or an access control graph. Typically they present *Views* to the user that provide visual feedback on the policy decisions. The different views are discussed in more detail together with the analysis of the case study in Section 10.5. Proprietary plug-ins can be used to visualise scenario dependent information. SPAT does not provide direct support for the visualisation of the scenario, as this is a highly domain-dependent issue.

Explanation Component The *Explanation Component* is actually a plug-in, however due to its importance in the analysis process it is depicted as a dedicated component in the architecture. The explanation component interacts with the other plug-ins to provide the user with a feedback for a selected policy decision. The explanation contains descriptions of the rules that have influenced the decision. This allows the user to query those policy

decisions that did not match his initial intent and quickly find the rules that require modification.

In the following we will present some of the modules in the context of a small case study.

10.3 Case Study

We present a small simplified scenario, that shows the use of our dynamic policy model for resource control. We will use this scenario in the subsequent sections, to illustrate the functionality of our analysis toolkit. The scenario is taken from the military domain and can be seen as a simplification of a typical bandwidth allocation problem in an adverse environment.

10.3.1 Scenario

A platoon is navigating an area, where long range communication is limited due to environmental conditions. The platoon consists of several small units and a command unit that carries a long distance transmitter. The communication within the platoon is enabled using short distance radio links. The quality of service of the long distance transmission is highly dependent on the environment the platoon is navigating. Dependent on the command units position there may be significant drops in the communication bandwidth or even areas where communication is not possible at all. The command unit is used to analyse and control the mission. It is constantly relaying mission related information back to the base and provides a relay service to the other members of the platoon. The access to the relay service is controlled by a policy with the following requirements.

1. All members of the platoon are allowed to relay information.

2. If the bandwidth is dropping below 50% then units that have not been involved in combat action within the last 2 time-units are denied to relay information.

3. If the bandwidth drops below 20% only the command unit can relay tactical and strategic information.

4. If the command unit is under attack, the units that are not in its direct proximity are denied to relay messages, regardless of the available bandwidth.

The scenario is graphically represented in Figure 10.2.

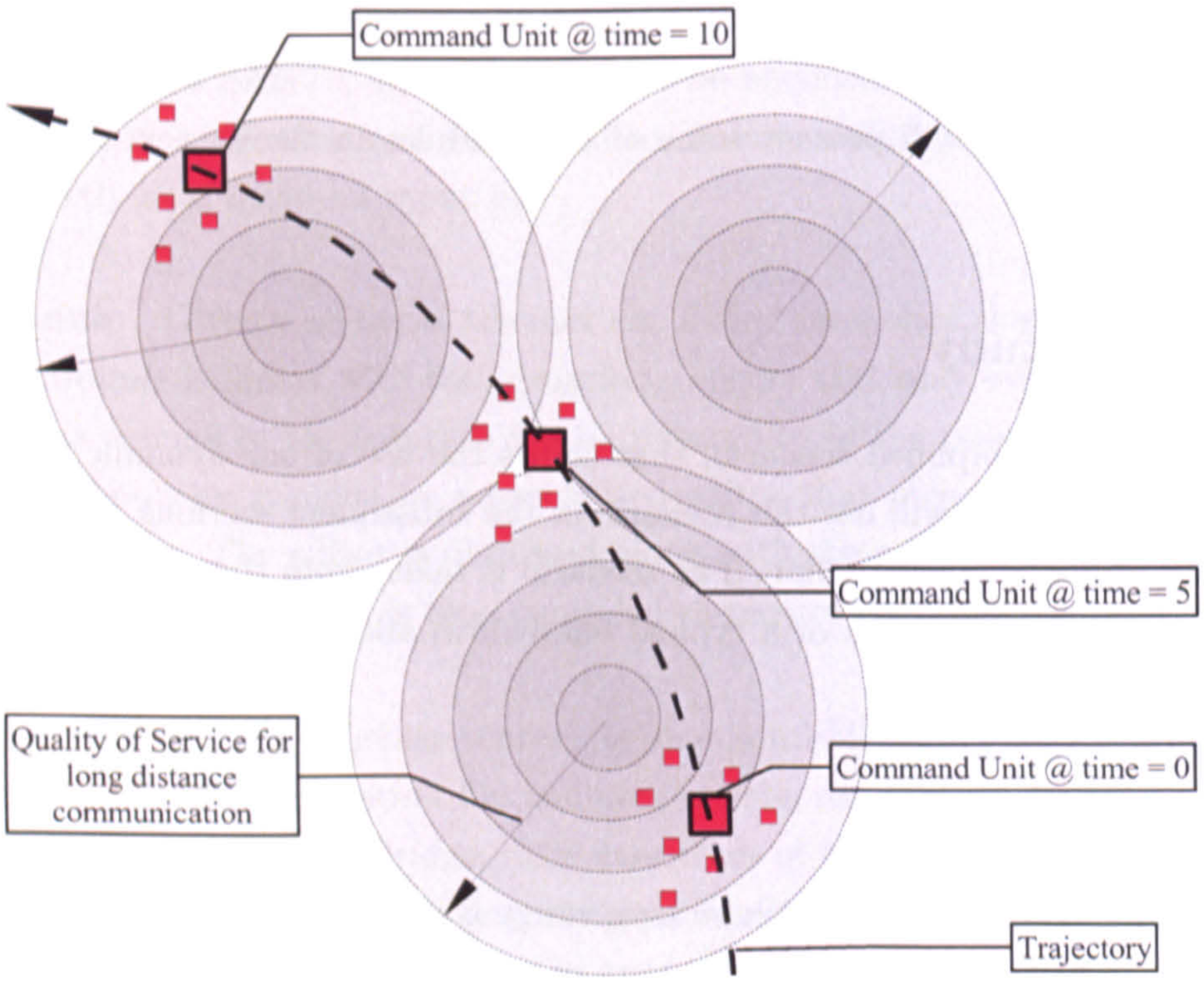


Figure 10.2: Platoon navigating a terrain

In Figure 10.2 the grey-shaded areas represent the different levels of quality of service in the terrain. The command unit and its subordinates move through the terrain along a predefined trajectory. The policy constraining the usage of the long range transmitter of the command unit is enforced by the command unit itself.

10.4 Policy Specification

In the following we will formalise the requirements individually as rules and then show how the rules can be composed to reflect the overall requirement specification. We formalise the first requirement as in Eq. 10.1.

$$member(S, platoon) \wedge command(O, platoon) \mapsto autho^{+}(S, O, relay) \tag{10.1}$$

Where *member* represents the membership relation between units and the platoon, and *command* the command unit relation. If S is a member of *platoon* and O is the command unit of the *platoon* it follows that S is authorised to *relay* information via the command unit O. The relations between units, platoon and command units are part of the scenario

specification for which the policy is to be analysed.

Whilst the first requirement uses only static information, such as the membership, the second requirement includes a temporal aspect. It takes into account the event of an attack on a unit over the last two time steps. This can be formalised, as in Eq. 10.2.

$$\begin{pmatrix} command(O, platoon) \wedge member(S, platoon) \wedge \\ 2 : (\Box(\neg\, combat(S))) \wedge \\ (\mathsf{true}\,;0 : (bandwidth(O) < 50)) \end{pmatrix} \mapsto\ autho^{-}(S, O, relay) \qquad (10.2)$$

If the bandwidth available to the command-unit O is now less than 50 (percent) and the requesting unit S has over the last two time units never been engaged in combat activity, then the unit is denied to relay messages. The formalisation of requirement 3 follows the same lines as requirement 2, without the temporal aspect.

$$\begin{pmatrix} command(O, platoon) \wedge member(S, platoon) \wedge \\ 0 : (bandwidth(O) < 20) \end{pmatrix} \mapsto\ autho^{-}(S, O, relay) \qquad (10.3)$$

Requirement 4 finally defines that if the command unit is under attack then units that are not in its proximity are denied to relay information, regardless of the bandwidth requirements stated in requirements 2 and 3.

$$command(O, platoon) \wedge member(S, platoon) \wedge \neg\, near(S, O) \mapsto\ autho^{-}(S, O, relay) \qquad (10.4)$$

The rule in Eq. 10.4 expresses the requirement partially. Additionally the requirement states that it overrides the requirements 2 and 3 and does only apply if the command unit is under attack. This can be seen as a dynamic change in the security policy, dependent on the event that the command unit is engaged in combat. We distinguish between two situations:

a) The command unit is not engaged in combat.

b) The command unit is engaged in combat.

In the first situation the simple policy that is applied consists of the rules (10.1), (10.2), (10.3) together with a decision rule that gives precedence to denials. The definition of the policy for the case that the command unit is not under attack is:

$$P = \left(\begin{array}{rcl} \begin{pmatrix} member(S, platoon) \wedge \\ command(O, platoon) \end{pmatrix} & \mapsto & autho^{+}(S, O, relay) \wedge \\ \begin{pmatrix} command(O, platoon) \wedge \\ member(S, platoon) \wedge \\ 2 : (\Box(\neg\, combat(S))) \wedge \\ (\mathsf{true}\,;\, 0 : (bandwidth(O) < 50)) \end{pmatrix} & \mapsto & autho^{-}(S, O, relay) \wedge \\ \begin{pmatrix} command(O, platoon) \wedge \\ member(S, platoon) \wedge \\ 0 : (bandwidth(O) < 20) \end{pmatrix} & \mapsto & autho^{-}(S, O, relay) \wedge \\ autho^{+}(S, O, A) \wedge \neg\, autho^{-}(S, O, A) & \mapsto & autho(S, O, A) \end{array} \right) \tag{10.5}$$

In the second situation, the policy definition does not contain the rules related to the bandwidth, but the rule that expresses the denial for units that are not in the proximity of the command unit.

$$Q = \left(\begin{array}{rcl} \begin{pmatrix} member(S, platoon) \wedge \\ command(O, platoon) \end{pmatrix} & \mapsto & autho^{+}(S, O, relay) \wedge \\ \begin{pmatrix} command(O, platoon) \wedge \\ member(S, platoon) \wedge \\ \neg\, near(S, O) \end{pmatrix} & \mapsto & autho^{-}(S, O, relay) \\ autho^{+}(S, O, A) \wedge \neg\, autho^{-}(S, O, A) & \mapsto & autho(S, O, A) \end{array} \right) \tag{10.6}$$

The change between the two situations is now expressed using policy composition:

$$R = (\langle combat(Cmd)\rangle P\,;\, [combat(Cmd)]Q)^{*} \tag{10.7}$$

The composition denotes that *unless* the command unit `Cmd` is under attack the policy P applies. On the event of an attack the policy changes to policy Q and applies as long as the command unit is involved in combat. The policy iterates between those two situations.

The advantage of this approach is that access requirements that are dependent on time and events, can be expressed at a higher abstraction level, without the need to explicitly encode the conditions in the premise of the rule. This leads to rules and policies that are easier to comprehend. Using policy composition, the policy designer can decide on the

time and event relations between different policies. SPAT provides a separate module for the specification of composed policies and rules. Figure 10.3 shows the prototype for the above policy specification.

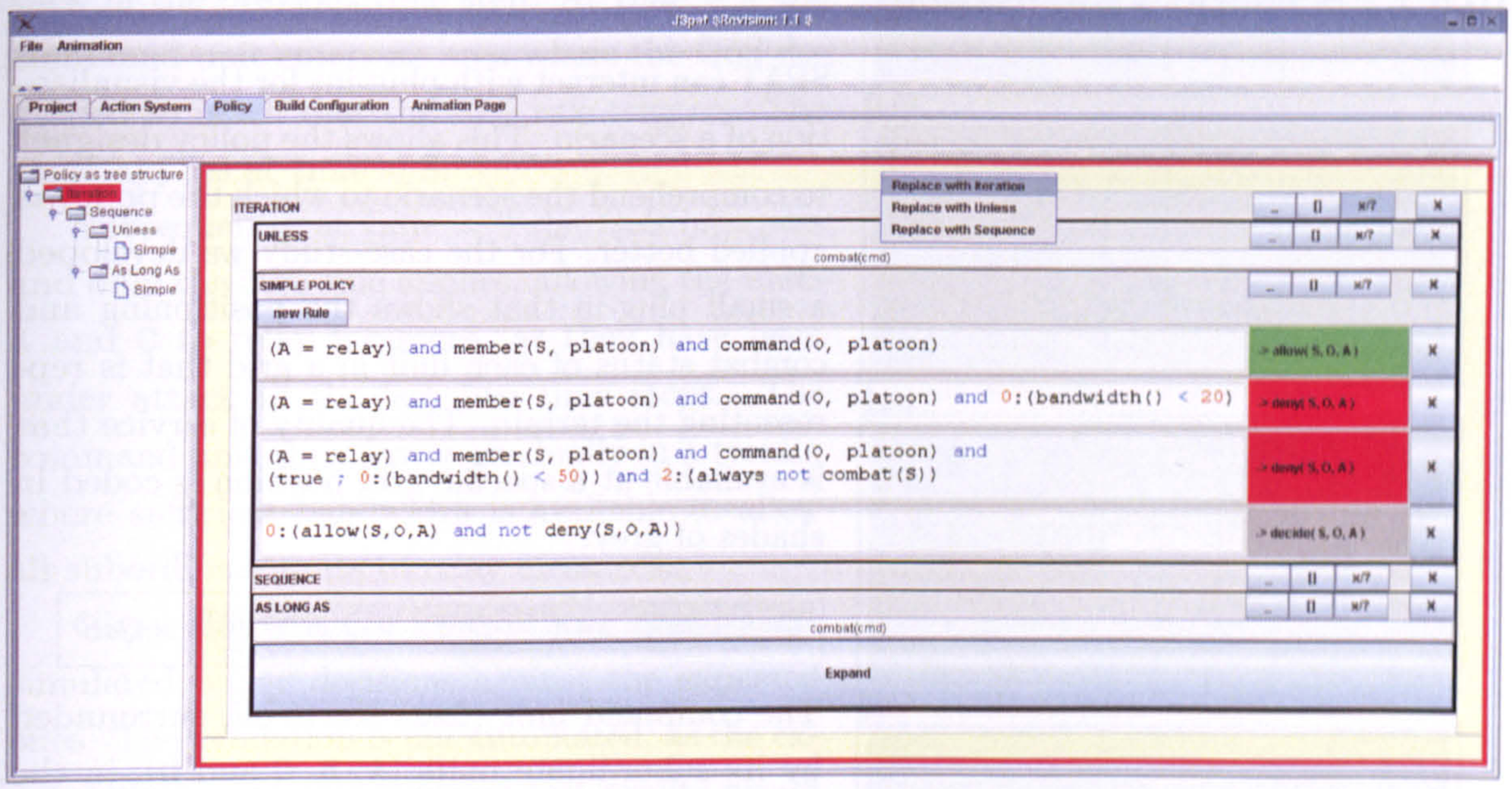


Figure 10.3: Prototype of the Policy Specification Module

The Policy Specification Module represents the policy composition as a structural *block* diagram. On the left *the structure is* represented as a tree; the right represents the corresponding block diagram. A simple policy is a list of *rules to which* rules can be added or deleted. The consequence of the rules can be changed by clicking on the *colour-coded* label.

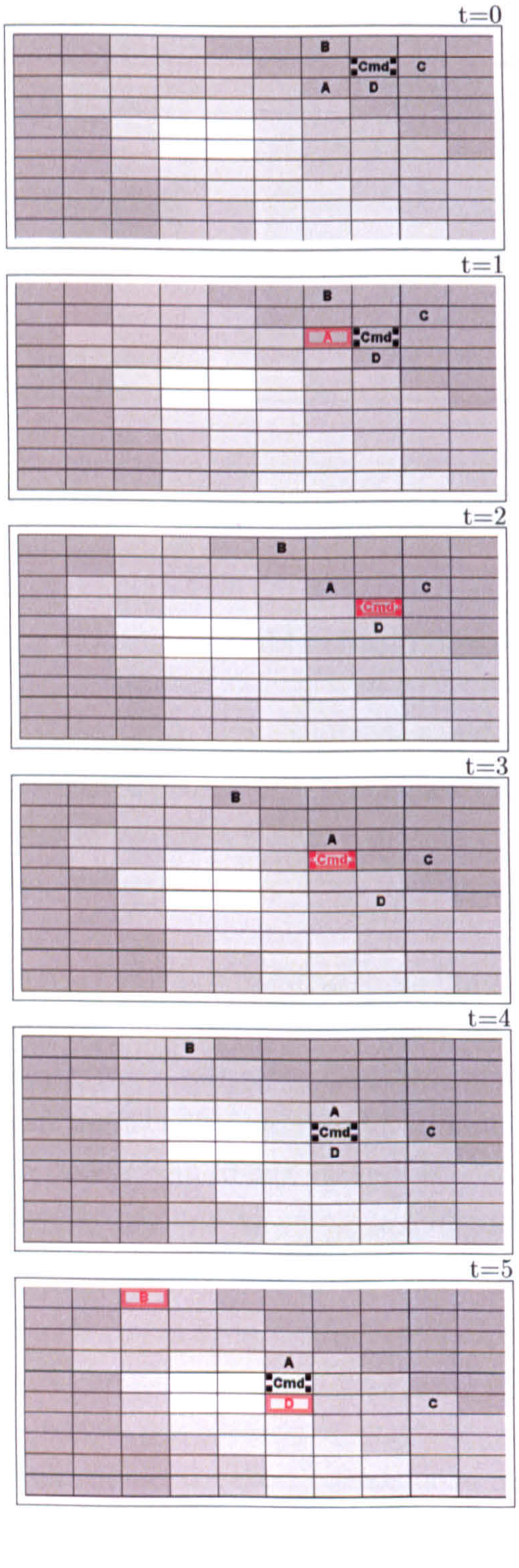
The specification module does also allow for the *hiding* of policies, that are currently not of interest. In the screenshot, the policy Q for the case that the command unit is under attack is hidden. It can be expanded again, by clicking on the corresponding *expand* button. Policies can be developed top-down and a bottom-up. The yellow frame shown in Figure 10.3 is used for a bottom-up approach to replace the framed policy with a composition construct. The specification module can also zoom in and out of policy compositions, by selecting in the tree-structure the appropriate policy component.

10.5 Policy Validation

Under policy validation we understand the process of analysing policy decisions in the context of a specific scenario. The validation is supported by plug-ins that visualise the

scenario and the policy decisions. Support for explanations how decisions have been made is also provided in form of a dedicated *Explanation component*.

10.5.1 Scenario Visualisation



SPAT can interact with plug-ins for the visualisation of a scenario. This allows the policy designer to comprehend the scenario to which the policy is applied better. For the case-study we developed a small plug-in that shows the positioning and combat status of each unit in a grid that is representing the terrain. The quality of service that is available at a specific grid position is coded in shades of grey:

$$QoS \geq 75\% \; > QoS \geq 50\% \; > QoS \geq 20\% \; > QoS$$

The command unit (`Cmd`) starts off, surrounded by its subordinate units (`A`, `B`, `C` and `D`), in the north-easterly corner of the grid and traverses diagonally down to the south-westerly corner. The event that a unit is under attack is visualised by the red frame surrounding the unit. For example at time = 1 the unit `A` is under attack.

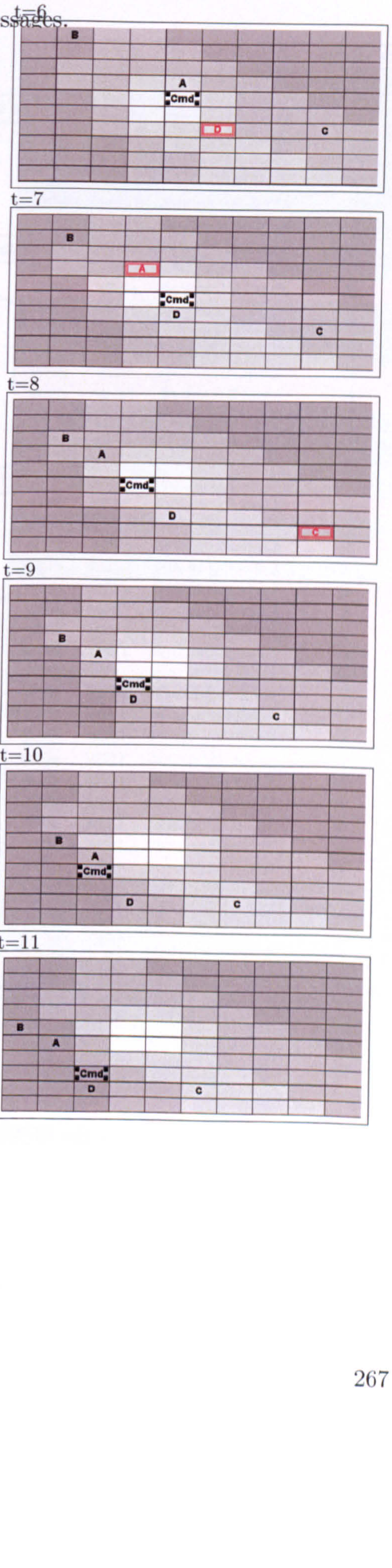
Initially the command unit traverses an area where there is sufficient bandwidth to allow all units to relay messages back to the base, however at time = 2 the command unit itself becomes involved in combat, triggering a change in policy that allows only nearby units to relay messages. We consider in this scenario units that are in a within a distance ($= \Delta x + \Delta y$) of two squares from the command unit to be near-by. At time = 5 the command unit enters an area where the quality of service degrades to less than 50 %. This means that the rule that denies units that have not been under attack within the last 2 time steps to relay messages applies. Consequently, only the units `B` and `D` should be able to

relay messages.

Importantly at time = 6 the unit B is still able to relay messages, as it has been under attack in the previous time step. At time = 7 the command unit enters an area where the QoS degraded so much that the 20% rule triggers. The same applies at time = 8.

Subsequently at time = 9 the QoS improves and only the 50% rule applies, allowing the units A and C to relay messages, as they have been under attack in the last two time steps. The command unit then moves gradually in a terrain where sufficient bandwidth is available to allow all subordinate units to relay messages.

Given the scenario we can now validate the simulated policy decisions against the expected ones. The validation is not automated, as the exact definition of the expected test-results would already represent an exact specification. The tool is intended to provide early feedback in the specification process itself and relies therefore on the interaction with the policy designer and his observations.



10.5.2 Policy Decisions

The policy decisions can be analysed using two different views. The first view is a tabular view that lists all access control related decisions that have been made during the simulation run. Alternatively only positive authorisations or negative authorisations can be displayed. The view currently supports the sorting by column and the filtering of the displayed information using regular expressions.

State ...	Subject ...	Object : CmdLRTr...	Action :...	Deci...
0	A	CmdLRTransmit	relay	allow
0	D	CmdLRTransmit	relay	allow
0	C	CmdLRTransmit	relay	allow
0	B	CmdLRTransmit	relay	allow
1	B	CmdLRTransmit	relay	allow
1	A	CmdLRTransmit	relay	allow
1	D	CmdLRTransmit	relay	allow
1	C	CmdLRTransmit	relay	allow
2	C	CmdLRTransmit	relay	allow
2	D	CmdLRTransmit	relay	allow
2	B	CmdLRTransmit	relay	deny
2	A	CmdLRTransmit	relay	allow
3	A	CmdLRTransmit	relay	allow
3	C	CmdLRTransmit	relay	allow
3	D	CmdLRTransmit	relay	deny
3	B	CmdLRTransmit	relay	deny
4	B	CmdLRTransmit	relay	allow
4	C	CmdLRTransmit	relay	allow
4	D	CmdLRTransmit	relay	allow
4	A	CmdLRTransmit	relay	allow
5	C	CmdLRTransmit	relay	deny
5	B	CmdLRTransmit	relay	allow
5	A	CmdLRTransmit	relay	deny
5	D	CmdLRTransmit	relay	allow
6	B	CmdLRTransmit	relay	deny
6	C	CmdLRTransmit	relay	deny
6	A	CmdLRTransmit	relay	deny
6	D	CmdLRTransmit	relay	deny
7	B	CmdLRTransmit	relay	deny
7	C	CmdLRTransmit	relay	deny
7	A	CmdLRTransmit	relay	deny
7	D	CmdLRTransmit	relay	deny
8	C	CmdLRTransmit	relay	deny
8	D	CmdLRTransmit	relay	deny
8	A	CmdLRTransmit	relay	deny
8	B	CmdLRTransmit	relay	deny
9	B	CmdLRTransmit	relay	deny
9	D	CmdLRTransmit	relay	deny
9	A	CmdLRTransmit	relay	allow
9	C	CmdLRTransmit	relay	allow
10	B	CmdLRTransmit	relay	allow
10	D	CmdLRTransmit	relay	allow
10	A	CmdLRTransmit	relay	allow
10	C	CmdLRTransmit	relay	allow
11	C	CmdLRTransmit	relay	allow
11	B	CmdLRTransmit	relay	allow
11	A	CmdLRTransmit	relay	allow
11	D	CmdLRTransmit	relay	allow

Figure 10.4: Tabular View of the SPAT Analysis Module

Figure 10.4 displays the access control decisions for the scenario. The view is sorted by states (abstract time) and filtered to only show the decisions made for the subjects `A`, `B`, `C` and `D` for the action `relay` on the long range transmitter of the command unit. For easier comprehension denials are colour-coded in red and allowances in green.

The policy decisions can then be compared with the scenario to increase the policy designers confidence in the specification. For example it is immediately clear that during the time that the command unit was situated in a very low QoS area all subordinate units were denied to relay messages.

10.5.3 Graphical Representation

For many users a graphical representation of access control policies is more appealing. This is for example the underlying assumption for LaSCO [68] policy specification framework. We believe that a representation as an access control graph is beneficial, especially for the analysis of *permissible* information flow [124]. SPAT provides a view of the filtered access control matrix as a directed graph, where the edges of the graph are labelled with the action names and states (color-coded: red for denial and green for allowance).

Subjects are represented as red triangles, objects as yellow triangles. The directed edges in the graph represent the actions that subjects can (or cannot) perform on the objects. The label at the edges describes the access control decisions. For the analysis of permissible information flow, all actions are categorised into *read* and *write* actions (see Section 3.3.2). The check whether there is a permissible information from one entity to another can be performed by analysing the connectivity in the graph.

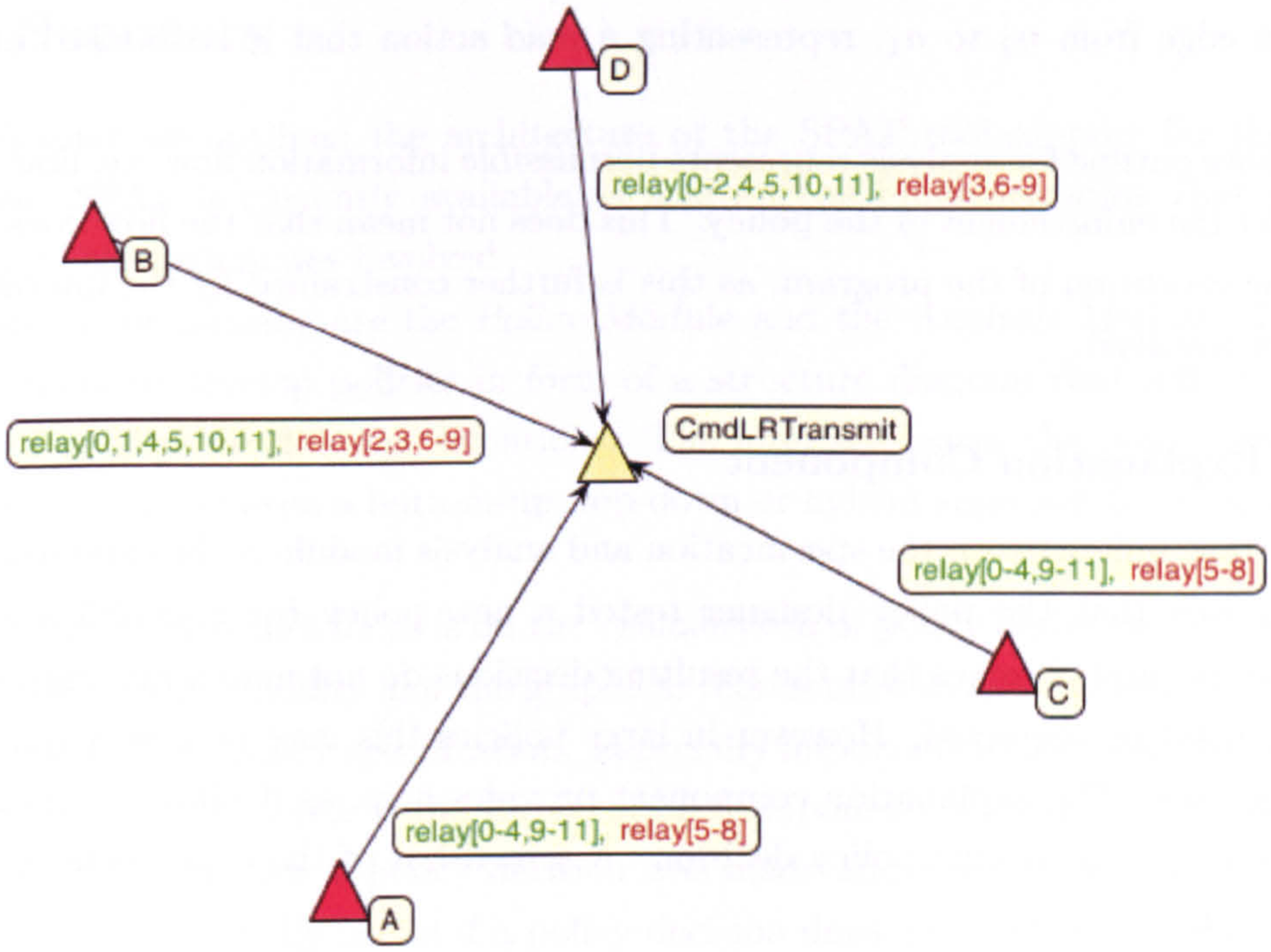


Figure 10.5: Graphical View

Direct Permissible Information Flow There is a direct permissible information flow from node n_1 to node n_2 at time t iff there is an edge from n_1 to n_2, representing a *write* action that is authorised at a time $t_w \leq t$ or if there is an edge from n_2 to n_1, representing a *read* action that is authorised at a time $t_r \leq t$.

Indirect Permissible Information Flow There is a indirect permissible information flow from node n_1 to node n_2 at time t iff there is a node n' for which there is a direct information flow from node n_1 to node n' at time t_1 and a indirect information flow from node n' to n_2 at time t_2 and $t_1 \leq t_2 \leq t$.

If we assume that the concrete implementation is known, we can take some possibility for covered channels that are part of the action execution into account. The actions can then be classified as *write*, *read* or *query*. A *query* action is a special form of *read* that is side-effect free, viz. the read does not affect the state of the object. The analysis is then along the same lines as explained above, however the definition of the direct information flow changes to:

Extended Direct Permissible Information Flow There is a direct permissible information flow from node n_1 to node n_2 at time t iff there is an edge from n_1 to n_2, representing a *write* or a non-*query read* action that is authorised at a time $t_w \leq t$ or if

there is an edge from n_2 to n_1, representing a *read* action that is authorised at a time $t_r \leq t$.

The above outline for analysis represents *permissible* information flow, i.e. flow that can occur under the enforcement of the policy. This does not mean that the flow does actually exist in the execution of the program, as this is further constrained by the interactions of the entities involved.

10.5.4 Explanation Component

The important link between the specification and analysis module is the explanation component. Given that the policy designer tested a new policy (or a modification in an existing policy) and observes that the resulting decisions *do not* match his original intent, the policy must be corrected. However in large policies this may be a very difficult and error-prone task. The explanation component provides help, as it allows to query which rules did influence a specific policy decision. A screenshot of the component is provided in Figure 10.6.

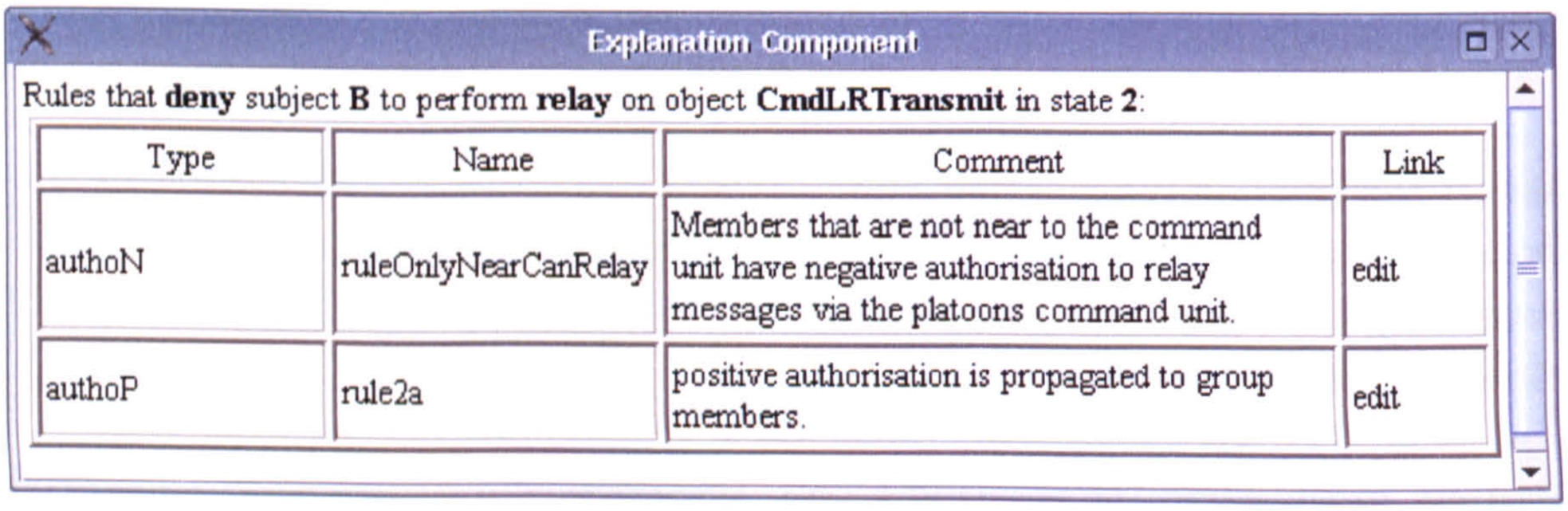


Figure 10.6: Explanation Component

The explanation component shown in Figure 10.6 was obtained by clicking on a the corresponding row in the tabular view shown in Figure 10.4. It is also available for the graph view on selection of an edge in the graph. The *Explanation Component* only displays the rules that lead to a decision in this context.

10.6 Summary

In this chapter we outlined the architecture of the SPAT tool-support for the analysis of policies. SPAT is currently available as a prototype implementation that shows the feasibility of the techniques involved.

The main components are the *Policy Module* and the *Analysis Module.* The *Policy Module* allows to develop policies in form of a structure diagram that reflects the composition of policies for different situations. The interface leaves the policy designer the freedom to choose between a bottom-up, top-down or hybrid approach to the specification of policies.

The *Analysis Module*'s focus is on the visualisation of policy decisions for the concrete scenario. The filtering ability and the graphical representation allow a user to concentrate on key aspects of the policy specification. Especially important to decrease the time in the specification – analysis cycle is the *Explanation Component.* This specialised component provides feedback on *how* a policy decision was made and which rules did influence the decision. This is especially useful if a policy decision does not capture the original intent of the policy designer.

Chapter 11

Conclusion

In this chapter we summarise our findings, highlight the contributions and critically review our approach to policy specification and enforcement. Additionally we outline future work to extend our contributions and address some of the criticism.

11.1 Introduction

In this chapter we summarise our work on the integration of functional, security and temporal requirements for the specification of secure Multi-Agent Systems. Section 11.2 reviews the main concepts that have been presented in this thesis in form of a short summary. We then highlight our contributions and compare them with previous and related work in Section 11.3. Although our contributions advance the research in policy-related areas of computing, our approach is not a silver-bullet that solves all problems. Consequently there is also criticism to our approach, which we compiled in Section 11.4. We conclude this chapter and the thesis in Section 11.5 with ideas that can improve the presented work in the future.

11.2 Summary

In this thesis we have shown how security requirements, that are expressed as dynamically changing policies, are enforced in an agent-based distributed system. The system's functional- *and* security requirements are defined together in form of a SANTA specification. The advantage is that security requirements are seamlessly integrated in the development process.

SANTA specifications define the system in terms of *agents* that are active entities acting on behalf of a user; *objects* that represent shared resources in the system; *policies* that describe protection requirements for both agents and objects; and *enforcement mechanisms* that provide the link between policies, agents and objects. An overview of these components and their interaction has been provided in Chapter 3.

The development process that is used to transform SANTA specifications into concrete, deterministic SANTA programs is based on the notion of refinement. The stepwise application of sound refinement rules makes the specification gradually more deterministic to a degree at which the specification becomes an executable program. Formal refinement is possible because SANTA has a sound specification-oriented semantics in Interval Temporal Logic (see Chapter 4).

Agents have been formally defined in Chapter 5 as entities that encapsulate variables and actions. The execution model of a single agent is staged in phases. After an *initialisation phase* the agent enters the Deliberation-Enforcement-Execution (DEE) cycle. The deliberation phase abstracts the decision making process of the agent. In this phase the agent prioritises its actions for execution. We do not impose a special mechanism for the decision-making and leave the developer the freedom to choose the appropriate agent-architecture.

Once the actions are prioritised the agent enters the enforcement phase. This phase

enforces a *behavioural* policy, viz. a policy that defines constraints on the agent's execution. We refer to an agent that enforces such a policy as a *vigilant agent*. The effect of the enforcement phase is that the original action prioritisation of the agent is modified to ensure the compliance with the behavioural policy. The policy can potentially overrule any decision the agent has made. This is especially advantageous if the agent's deliberation is complex and uses third party modules or AI learning techniques that are difficult to test and verify before the agent is deployed.

In the execution phase the agent executes either a *local action*, that is an action that performs only local computation, or a *remote action* that invokes an interface of a shared object. If an agent is terminated, it leaves the DEE cycle and enters the *termination phase*.

Objects represent resources in the shared environment of agents. Agents can access objects through interfaces to modify the variables that are encapsulated within the object. We formally defined objects in Chapter 6. The access can be restricted in *environmental policies*, that are either vigilantly enforced by an object, or centrally by a *security enforcer* for a collection of objects.

To protect the access to object interfaces and to constrain the autonomy of agents we formally define policies in Chapter 7. Policies are composed of rules that capture history-dependent *authorisation*, *delegation*, *obligation* and *integrity* requirements. Policies can be composed along a temporal axis. In this case the component policies capture the requirements that apply in a specific situation. The composition operators then define when the transition between the situations takes place. Policies can also be composed along a structural axis, viz. the component policies define the requirements for a specific *scope* (e.g. a department) and the composition operator defines how conflicts between the components are resolved.

Policies are defined at a different abstraction level than the system itself. While the system defines the transition from one state to another at the level of concrete assignments to variables, the policy is defined in terms of executed actions or invoked interfaces. This is a much coarser grained view. The concrete mapping between policy abstraction and system implementation is defined by the enforcement mechanisms using *temporal projection*. We detailed the two vigilant mechanisms, where the enforcement is part of the agent or object itself, and the centralised *security enforcer* mechanism in Chapter 8.

In Chapter 9 we have shown how specification constructs such as the temporal assignment can be refined into concrete implementable code. Important for the success of a policy language is its enforceability, that is the development of enforcement mechanisms that can ensure that the requirements expressed as a policy are not violated by the system. In Chapter 9 we have therefore also shown how policies can be transformed into

a deterministic normal form and how agents can be refined to implement corresponding enforcement code.

Finally we provided tool-support for the analysis of policies, where we focussed on the validation of high-level policies. The use of the tool for a specific case-study has been detailed in Chapter 10 and [75]. The prototype of SPAT allows for the specification of a policy that is tested under a given scenario. The resulting policy decisions are visualised and the policy designer can compare the results to see whether they meet the informal requirements. The approach is similar to early prototyping in traditional software-development processes and geared especially towards policies. Most important is the feedback that is readily available to the policy-designer on which rules did influence the observed policy decision.

11.3 Contributions

The main contribution of this work is the integration of functional and security and temporal requirements in a uniform and formal framework to allow for the analysis of the effects that policy decisions have on the system behaviour. Secondary contributions are extensions to the policy model proposed in [124]; the abstraction of policies using temporal projection; and the refinement of policies into concrete enforcement code. We detail the contributions in the following.

11.3.1 Integration of Functional and Security Requirements

The motivation of this work is the realisation that security requirements must be addressed early in the system development life-cycle and cannot be bolt-on as an afterthought. This has been widely accepted in the community and lead to the integration of security in for example UML [79]. However, we believe that for applications in which security is a *critical* factor, viz. the violation of security requirements may lead to loss of life, a more rigorous approach must be taken. Applications where security is critical are often found in governmental information systems that process sensitive data of a large number of citizens, or in the military domain where confidentiality is critical to the success of a mission.

The aim of this work was to integrate functional, security and temporal requirements for the development of distributed systems in a uniform and formal framework. We chose Multi-Agent Systems as a representative of distributed systems, because the autonomy and encapsulation of the different system components appeared to be a good candidate to model applications in the before mentioned domains [19]. We have provided in this work a rich computational model for a Multi-Agent System in which a) the autonomy of agents can be limited and b) the access to shared resources is controlled by policies.

The model does not constrain the agent-architecture that is used for the implementation of the individual agents in the MAS.

The policy model is based on dynamically changing policies that have been introduced in [124] and previous work [74]. It provides a flexible and compositional approach to the specification of history-dependent policies that can dynamically change over time or on the occurrence of events. The model has a formal semantics given in Interval Temporal Logic. ITL is a suitable foundation for the model as reasoning over temporal aspects of a specification is natural. It also provides a compositional proof-system [101] that improves the scalability of the approach.

The SANTA language integrates the specification of functional requirements in form of a MAS, security requirements in form of policies and links them through (de-) centralised enforcement mechanisms. The semantics of all these components is given in ITL allowing for the uniform analysis of security properties and their interaction with the functional part of the system. This is a contribution in itself, as most other work either dedicates itself to the specification of agent-systems (e.g. [58]) or policies (e.g. [72, 20]). Often the link between the two is not clearly defined and the enforcement mechanisms are programmed in an ad-hoc manner. In the framework presented here the relation between the system and policies is made explicit by the abstract definition of enforcement mechanisms (see Chapter 8) and their refinement (see Chapter 9).

We believe that the SANTA framework helps to close the gap that exists between the specification of security policies [114] and their enforcement in concrete mechanisms by providing a development approach that allows to derive these mechanisms from their high-level specifications.

11.3.2 Extensions to the original Policy Language

The original model [124] has been extended to allow for the specification of *obligations* and *integrity constraints* as well *scoping* and *parallel composition* of policies.

Obligations Obligations are an important aspect of management policies, as they define what a subject *must do* as opposed to access control which defines what a subject *can do*. An obligation is for example that an agent must enter a transaction-log to a central database or that a user must revoke a delegated right after a certain time. Similar requirements can also be found in for example the BMA policy framework [6]. This type of requirements could not be expressed in the original policy language presented in [124]. Especially as the enforcement of obligations is highly dependent on the system itself.

Other policy languages (e.g. [108, 120]) take the view that obligations are directly associated with the execution of an action. We take here a more liberal view in that

we define an obligation to be dependent on any observable condition or event. We also take the view that *pre-obligations*, e.g. *the clinician must obtain the consent of the patient before passing information on to others*, are actually authorisations in disguise. The above example could be rephrased to *The clinician cannot pass information on to others if he did not obtain the patient's consent before.* In for example XACML an obligation is actually an obligation of the enforcement mechanism, viz. the enforcement mechanism must perform a certain action, rather than being associated with a subject. Although this is not fundamentally different it provides more flexibility in the specification.

Integrity Integrity rules define constraints on the observable effect of an action execution and allow to discard results that do not meet these constraints. Integrity policies for a single action/interface invocation are well known and are nowadays supported by many programming languages (e.g. Java [63]) in form of *assertions* [65]. The benefit of our approach is that these assertions can also take the history of the execution into account. We provided an example for the integrity of the computation of Fibonacci-Numbers in Chapter 7. The history of the execution is at the policy specification level not explicitly defined, but is captured in the semantics of policy rules. The advantage is that the additional variables that would only be used to check the integrity of the execution are not implemented as part of the program itself. This honours one of the key-principles in software testing stating: *do not include functionality required only for testing in the original code.* The benefit of this approach has also be noted in [43], but seems not to have been a major concern in the development of policy languages.

By defining integrity policies, one specifies the expected behaviour of a component at a high level of abstraction. with respect to the testing community this specification is desired to be *precise*, viz. the actual behaviour must not deviate. Testing is concerned with the finding of software faults before deployment. With policies we a concerned with the validation of properties in the deployed and running system. One of the applications, especially for open systems, is the establishment of trust-relationships between previously unknown entities. In this case the integrity policy defines a set of *acceptable* behaviours, that when matched by the actually observed behaviour lead to the establishment of trust. Key to this use is that the violation of an agent's integrity policy on an object's interface execution *does not affect* the object. For example when booking two seats in Albert Hall, my agent can reasonably expect that the available seats for the second booking do not contain the seat that was reserved for me in the first booking. If this is violated my trust in the booking system will decrease (even if I received an adequate confirmation for the first booking).

Scoping and Parallel Composition The advantage of defining policies that can change over time and events in a compositional fashion is that it allows for a modular approach to policy specification. Policy composition has previously been addressed in many languages e.g. [72, 138, 46, 108, 18, 14], however they do not address composition along the temporal axis. We feel that this form of composition is important, as it allows the policy designer to not only partition the task of policy specification in departments, but also according to *situations*. This is especially suitable for policies in work-flow based systems that are naturally divided into stages. Other authors [28, 17] did also recognise the need for temporal dependencies of authorisations and allow for the expression of periodic constraints e.g. "between 10:00 and 14:00 o'clock every Tuesday". Our approach is more flexible in that sequences and order relation can be specified without explicitly defining time.

The difficulty of introducing scope and parallel composition is the dynamics of the model. One of the key advantages of the model presented in [124] is that policies are conflict free, viz. the outcome of a policy decision is always defined. However, by introducing parallel composition new conflicts can be easily introduced, especially when the component policies do change dynamically. We have shown in Section 7.5.2 that the policy composition using a simple conjunction does not adequately capture the intuition. We proposed a composition that *hides* the effect of the component policies and allows additionally the specification of an explicit conflict resolution. This approach differs greatly from the traditional set-based approaches for policy composition as it captures the dynamics of the component policies.

11.3.3 Policy as an Abstraction

In [124] policies have been defined at the same level of abstraction as the system. This lead to difficulties in matching the semantics of a change in policy on a specific event. For example by executing an action that assigns `x:=0;x:=1;x:=2` and a policy that changes on `x=1` and again on `x=2`, it is not clear how these events can a) be observed and b) be implemented. With the proposed abstraction of policies we clearly define the observable states and events that can affect the policy (see Chapter 8).

Additionally the implementation of more complex enforcement mechanisms, that for example take the history of the execution into account, could not be implemented because their implementation would affect the semantics of the policy itself. By defining policies at a higher level of abstraction and showing how the enforcement maps between these abstraction levels we are now able to implement more sophisticated enforcement mechanisms. An additional benefit is that it allows for *delayed refinement*, viz. policies and actions/interfaces can be refined independently. This was detailed in Chapter 8 and 9.

11.3.4 Enforcement

Key to the success of a policy language is its enforceability. Although [124] addresses the implementation of policies in Secure Action Systems (SAS), only a small subset of policy rules could be enforced. The premise of the rules was restricted to ITL state-formulae, viz. formulae that do not contain any temporal dependencies. This is an obvious drawback, as the power of the policy rules stemmed from the fact that history-dependencies could be expressed naturally as ITL formulae. We extended the class of rules that is enforceable to a negation-free subset of ITL with bounded quantification and have shown how the corresponding enforcement code can be formally derived (Chapter 9).

The dynamic change of policies was implemented in [124] by composition of SAS, using similar operators as for policy composition. It was then shown how the SAS can be brought into a normal form that enforces the changing policy. In this work we chose a different approach and use compositional refinement to implement the policy change.

We also relaxed the assumption of a single, centralised enforcement mechanism and allow for the distribution of policies to different enforcement mechanisms. Although this restricts the dependencies of policies to the part of the system that is observable by the enforcement mechanism, we feel that it is closer to actual system implementations.

Enforcing Dynamically Changing Policies The enforcement of dynamically changing policies has an advantage over the enforcement of other policy languages in which different situations are expressed as additional constraints on the policy rules to guarantee that the rule does only apply in the right context. For languages, in which policies do not change dynamically, the set of rules that must be evaluated by the enforcement mechanisms at any point in time remains constant. The specification of dynamically changing policies carries additional information on the *transition* between phases. This enables the enforcement mechanism to recognise the transition and to evaluate only those rules that apply in the current situation. Depending on the number of situations that can be distinguished this can lead to substantial savings in the time that is needed for the enforcement of a policy.

For example XACML [108] provides checks for the applicability of a policy. This is also aimed at the reduction of the computational effort that is needed to determine policy decisions. Still, the applicability of a policy is checked with every access request.

Deriving the Required History Unlike other models, e.g. [72], the enforcement of our policy model does define how policy related events and states are maintained in form of histories. We have shown how the process can be optimised to maintain only the information that is needed to make policy decisions. This eliminates the need for *meta-*

policies that define the purging of a history data-base as proposed in [61]. The possibility to optimise the history in this sense has been recognised in [118], however it is not based on a sound mathematical foundation. The benefits of this approach have also been highlighted in [43] where they are applied to history dependent integrity checks for data-base systems that are defined as past time temporal logic formulae.

Generation of Enforcement Code The refinement of policies into concrete enforcement code can be automated. We presented a prototype of such a compilation mechanism in Chapter 9. This allows for the generation of optimal enforcement code for a specific policy during run-time; viz. if a policy is changed, the new and updated enforcement code can be created. The advantage of this approach in comparison to a logic-programming based approach to policy evaluation is that a detailed timing-analysis can be performed on the enforcement code. Given a concrete execution platform real-time guarantees on the policy decision making can be provided.

Temporal Requirements More recently with the development of Usage Control Models (UCON) [145] the benefit of temporal logic for the specification of policies has been adopted. However, their approach seems to mainly focus on the maintenance of system controlled attributes rather than the specification of history dependencies of policies using temporal logic. Also their approach does not address the composition of policies, concurrency of usage requests and the enforcement of policies.

Related work in the Web-Service/Agent community to adjust the autonomy of agents, e.g. [134], use a description-logic based policy language. It is not clear how one can reason within these frameworks about temporal dependencies of policies without explicitly encoding time. In [136] AND-OR trees are used to model constraints on an agent's goal-adaption – again this does not seem to be suitable for the specification and analysis of temporal requirements.

11.4 Critical Remarks

In this section we compiled some of the criticism that we encountered when presenting our work to the community in form of papers and presentations.

Another Policy Language – why? A commonly made critique is that there are already many policy languages out there, why not take one and extend it. One paper review for example suggested to use ITL to formalise XACML, as it has similar concepts of conflict resolution between policies (as does e.g. [72]). In response to this critique we agree with Becker et.al. who state in [20]:

> Languages such as XACML, XrML, or SPKI/SDSI ... are specified by a combination of lengthy descriptions and algorithms that are ambiguous and, in some cases, inconsistent. Post-hoc attempts to formalise these languages are difficult and reveal their semantic ambiguities and complexities ...

The aim of this work is to provide a uniform framework that integrates functional, security and temporal requirements. In the achievement of this aim the overhead of formalising many of the complex and sophisticated concepts found in industrial policy languages is not justifiable — especially if the result is that the formalised language is ambiguous or not consistent and additional effort would have to be spent in the rectification of these problems.

We clearly see the benefit of using an expressive formal model to capture the semantics of a widely adopted language with exactly the purpose of improving the language. However, this was not in the scope of this thesis. Additionally we feel that the concept of dynamically changing policies is important for policies in domains that are characterised with a high degree of uncertainty and that need to cope with highly dynamic situations. These concepts have not been integrated in other policy languages.

Policies should be interpreted! A commonly taken view is that policies are data that represents an input to an algorithm (e.g. the Policy Decision Point (PDP), see Section 2.3.3) that decides then on the access control decision. We take a different view in that the policy is for us an integral part of the PDP. In terms of flexibility, viz. changing the policy during the system execution, we would argue that it is as difficult to redirect PDP-requests to a different (new) PDP as it is to change the policy of a single PDP. Especially in interpreted languages, such as Java, the difference is merely conceptual.

The advantage is that during the compilation certain optimisations can be performed — a policy interpreter on the contrary would be implemented in a way that allows the enforcement of all possible policies. The benefits are for example discussed in [111, 18] that dynamically induce the enforcement code in existing programs.

Why can Obligations only be vigilantly enforced? We take the view that an agent (subject) is *obliged* to take an action. Given the autonomy of an agent, viz. it is in control of its actions, the decision to execute an action must be made as part of the agent's behaviour. Only a vigilant agent can alter the decisions made in the agent's deliberation phase, due to autonomy and encapsulation.

However, as we mentioned before other work e.g. [108] takes a different approach and sees obligation as obligation of the enforcement mechanism. If we would consider a security enforcer as being an active entity of the system, then of course this type of obligation could

also be enforced. This is analogous to the real world: it is difficult to *force* somebody to take a specific action. Typically it is only enforceable by imposing a form of punishment for not compliance. The process of punishing, however, is then undertaken by a different (active) entity, i.e. the enforcer. We consider to extend the notion of a security enforcer in *future work* (see Section 11.5).

The language is complex and real systems are difficult to express? Actually, the language itself is comparatively simple in its semantics, especially when compared with object-oriented systems or policy frameworks that make use of classes, inheritance and instantiations. This critique applies more to the *syntax* of the language, that is admittedly not overly appealing to system engineers.

Two points must be emphasised in response to this critique. Firstly, languages to specify the system (e.g. Java) and the system to specify the policy (e.g. Java's Policy Files) are typically seen as two distinct languages. Obviously each of them seen individually is less complex. The disadvantage is however, that relations between the two languages are not clear. Secondly, with respect to to the policy language the security requirements that can be expressed in a single rule are complex and their enforcement is non-trivial.

For the example that was used in Section 9.4 to show the refinement of a vigilant agent the resulting behaviour of the system was indeed complex, albeit the system description and the policy specification alone where short and concise. Contrary to the claim that it is undesirable to have such complex behaviour emerging from the enforcement of policies we argue that the integration of security and functionality within the same framework highlights the complexity that failures induced by enforcement mechanisms create in a system. We think it is important to highlight the effect that policy enforcement has on the system execution and argue that these complexities are already present in today's systems. By keeping the discussion of policies, system and enforcement separate they are not addressed by most of today's research on policies.

The structure of the system is too static We agree that by assuming a finite and static set of subjects, objects and actions in the system we lack the flexibility of today's systems to dynamically create new entities. We would like to address this issue in future work by catering for classes and instantiations of classes with respect to agents, objects, policies and enforcement mechanisms.

Formally, however, we can assume that due to finite resources in a software system the number of agents and objects will be finite. Therefore we can assume that the instantiation of an entity can be modelled by "activating" an existent, but unused entity. In this case defining policies in terms *of types (for objects)*, groups and roles (for subjects) or MLS

approaches helps to control the dynamic instances, as they abstract the policy specification from the concrete identities.

Formal Methods typically do not scale well Formal development methodologies have always been suspicious from the main-stream software development point of view. This is mainly due to the difficulty in their application, e.g. especially trained personnel is required, and the lacking capability to apply them to large-scale projects. In this work we aim to address software applications that are security-critical, i.e. that require a high level of assurance that security requirements are not violated. This demands a more rigorous and formal approach.

We feel that the approach that was presented in this work is in terms of scalability workable even for larger software projects. Firstly, the design of agents is modular, viz. to be able to implement the security mechanisms it is not necessary to refine the deliberation phase, or the statements that are executed in an action. Secondly, the policy model is specially suitable to express large and complex requirements due to its compositionality. When defining a policy one can focus on a particular aspect of the policies e.g. a single department for structural composition or a single situation for temporal composition. We feel that this greatly increases the ability of policy developers to comprehend and maintain large policies.

From a verification point of view the underlying logic has also the advantage of being compositional. This means that properties of the whole system can be inferred from properties of its components. This further supports the scalability of the approach.

11.5 Future Work

In this section we address some of the concerns that have been raised previously and highlight areas of research which we feel are worth pursuing in the future.

Active Security Enforcer

The *security enforcer* that has been introduced in Chapter 8 is based on the semantics of an object. It represents a passive entity that mediates the access to the protected objects. The passiveness of the enforcer represents a drawback when obligations should be associated with an enforcement mechanisms. One way of addressing this is by defining the security enforcer as an *active* entity.

In this case the security enforcer would extend the semantics of agents and objects. The ability to execute actions would enable the enforcer to comply with obligations. Interfaces allow for the provision of administrative functions, such as role-assignment, delegation,

etc. to other agents in the system. Another benefit would be that networks of security enforcers can exchange information on events, reputation or similar.

Models for Event-Distribution In natural extension to the *active security enforcer* it is conceivable to create *models for event-distribution*. This would allow the enforcement of policies that depend on events which are not directly observable by the enforcement mechanisms. These models would have to guarantee that events are distributed timely, viz. an access control decision is not made based on incomplete information (or if it is can be revoked whilst the access is ongoing — similarly as conceived in UCON).

SPAT towards an industrial strength tool

The prototypes for analysis and automatic enforcement code generation require attention. For SPAT it seems to be increasingly important to appropriately link between the specification and analysis modules. Additionally consistency checks, like the once that are made during the enforcement code generation provide valuable information to the policy designer, viz. if a policy is not satisfiable it is likely that the designer was mistaken in his conception of the rule. We are also investigating how a propositional subset of the policy rules could be model-checked using for example PITL2MONA [62].

Language extensions

The SANTA language provides in its current state relatively low-level concepts for the specification of agents and objects. We are interested of extending the language to provide well-known concepts like classes and inheritance as well as high-level communication constructs for inter-agent communication. The latter is based on using objects as channels or message-boxes, depending on the target platform.

The concept of classes, instantiation and inheritance can then be used to develop a library of policies and enforcement mechanisms. For example it would be desirable if enforcement mechanisms that support RBAC (i.e. that provide interfaces for role-assignment/activation) can be predefined and included in new system developments.

Provide an alternative formalisation for e.g. UCON or XACML

To address the concern of why we would want to create another policy-language we aim to show how UCON or XACML policies could be expressed using our framework. This could potentially detect ambiguities in these languages and thus help to improve them.

Trust: How to deal with the Unknown

Finally we want to exploit the developed concepts to model trust relationships between entities based on their past (observable) behaviour and the risk that is associated with an interaction. We aim to extend the framework in such a way, that entities can establish trust through experience and are able to share this information with other entities. This does obviously require a richer model for the decision-making of agents than it is currently present in SANTA. This line of investigation will be sponsored by the DIF-DTC consortium as Project 7.6 "Trust Management in Collaborative Systems". An outline of the proposed Trust Management Framework is depicted in Figure 11.1.

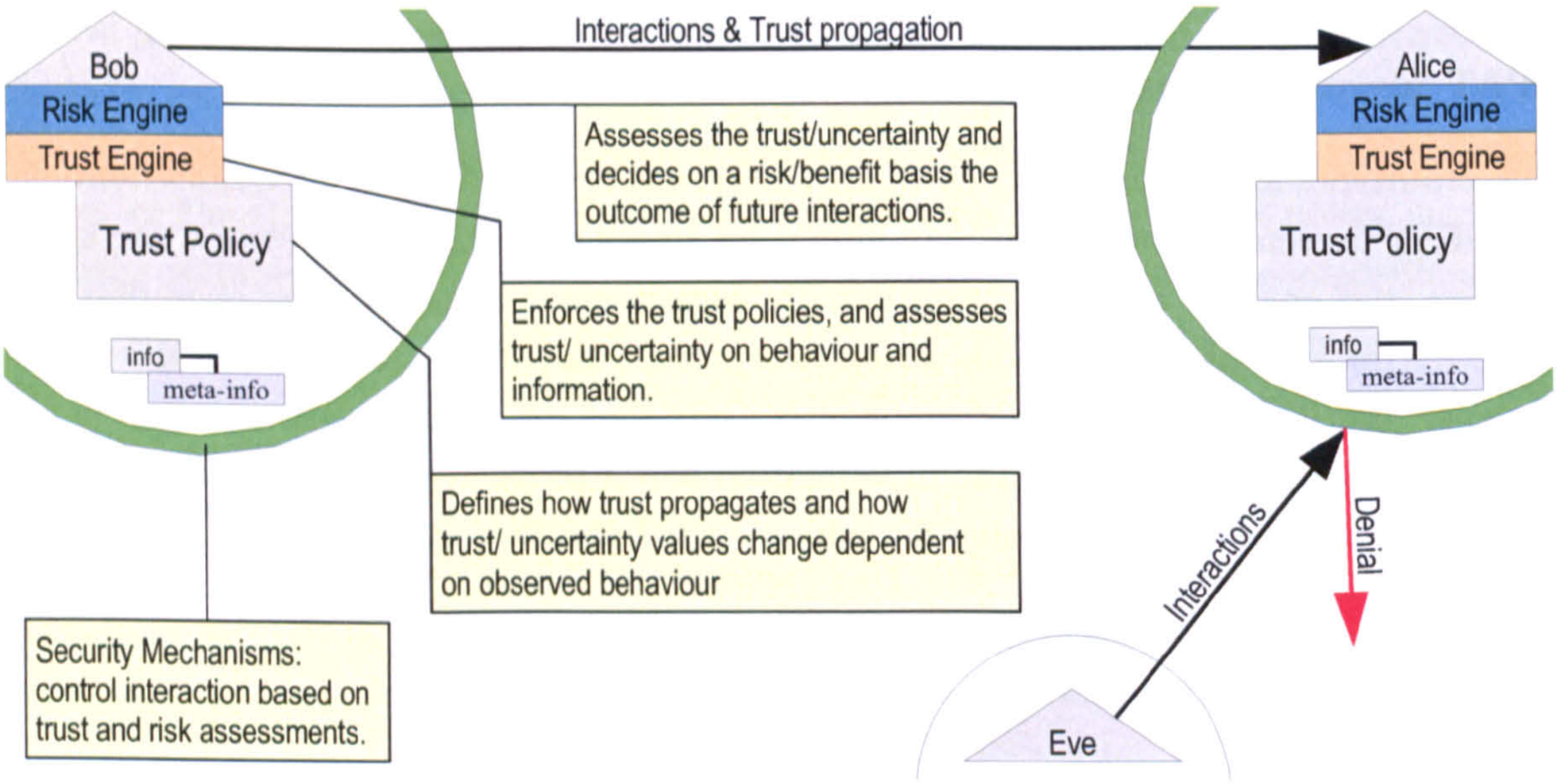


Figure 11.1: Outline of the Trust Management Framework

In this framework a trust-policy defines how an entity establishes trust and how it participates in the trust-establishment process of other entities through trust-propagation. The framework links between the subjective notion of trust and the actual security mechanisms that have been the focus of this work. Central to the approach is also the assessment of risk, that is the risk involved in engaging in an interaction with another entity or the risk involved with basing decisions on information that was provided by other entities.

References

[1] Martín Abadi. Logic in Access Control. In *Proceedings of the 18th Annual Symposium on Logic in Computer Science (LICS'03)*, volume 15, pages 228–233, Ottawa, Canada, June 2003. IEEE Computer Society Press. Available from: `citeseer.ist.psu.edu/article/abadi03logic.html`.

[2] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A Calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993. Available from: `http://doi.acm.org/10.1145/155183.155225`.

[3] Martín Abadi and C. Fournet. Access control based on execution history. In *10th Annual Network and Distributed System Symposium (NDSS'03).*, February 2003. Available from: `citeseer.ist.psu.edu/abadi03access.html`.

[4] S. Aljareh and N. Rossiter. Toward security in multi-agency clinical information services. In *Proceedings of Workshop on Dependability in Healthcare Informatics Edinburgh*, pages 33–41, March 2001. Available from: `citeseer.ist.psu.edu/aljareh01towards.html`.

[5] Anne Anderson. A Comparison of two Privacy Policy Languages: EPAL and XACML. Technical Report SMLI TR-2005-147, Sun Microsystems, September 2005. Available from: `http://research.sun.com/techrep/2005/smli_tr-2005-147/TRCompareEPALandXACML.html`.

[6] R. Anderson. Information Technology in Medical Practice: Safety and Privacy Lessons from the United Kingdom, 1998. Available from: `citeseer.ist.psu.edu/anderson98information.html`.

[7] Ross Anderson. Security in clinical information systems. Technical report, for British Medical Association (BMA), Computer Laboratory University of Cambridge Pembroke Street Cambridge CB2 3QG, January 1996. Available from: `http://www.cl.cam.ac.uk/~rja14/Papers/policy11.pdf`.

[8] Ross J. Anderson. *Security Engineering, A Guide to build dependable Distributed Systems.* John Wiley & Sons, Inc., New York, NY, USA, 2001. Available from: `http://www.cl.cam.ac.uk/~rja14/book.html`.

[9] ANSI (American National Standards Institute). ANSI INCITS 359-2004 Role Based Access Control. Technical report, American National Standards Institute, 25 West 43rd Street, New York, NY 10036, February 2004.

[10] M. Archer and E. Leonard. Analyzing Security-Enhanced Linux Policy Specifications. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, 2003. Available from: `http://chacs.nrl.navy.mil/publications/CHACS/2003/2003archer-policy03.pdf`.

[11] R.J.R. Back. Refinement Calculus, Part II: Parallel and Reactive Programs. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes In Compter Science*, pages 67–93, 1989.

[12] R.J.R. Back and R. Kurki-Suonio. Distributed Co-operation with Action Systems. *ACM Transactions on Programming Languages and Systems*, 1984.

[13] R.J.R. Back and J. von Wright. Refinement Calculus, Part I:Sequential Nondeterministic Programs. In J. W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes In Compter Science*, pages 42–66, 1989.

[14] M. Backes, M. Durmuth, and R. Steinwandt. An Algebra for Composing Enterprise Privacy Policies. In P. Samarati, D. Gollmann, and R. Molva, editors, *Proceedings of 9th European Symposium On Research in Computer Security (ESORICS)*, pages 33–52, September 2004.

[15] Gene Ball, Dan Ling, David Kurlander, John Miller, David Pugh, Tim Skelly, Andy Stanskosky, David Thiel, Maarten Van Dantzich, and Trace Wax. Lifelike Computer Characters: The Persona Project at Microsoft. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 191–222. AAAI Press / The MIT Press, 1997.

[16] Arosha K. Bandara, Emil C. Lupu, and Morris Sloman. *Handbook of Network and System Administration*, chapter Policy Based Management. Elsevier, to appear in 2007.

[17] Steve Barker and Peter J. Stuckey. Flexible Access Control Specification with Constraint Logic Programming. *ACM Transactions on Information and System Security*, 6(4):501–546, November 2003.

[18] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM Press.

[19] P. Beautement, D. Allsopp, M. Greaves, S. Goldsmith, S. Spires, S. Thompson, and H. Janicke. Autonomous Agents and Multi–Agent Systems (AAMAS) for the Military - Issues and Challenges. In Robert Ghanea-Hercock and Mark Greaves and Nick Jennings and Simon Thompson, editor, *Proceedings of 1st Workshop on Defence Applications for Multi-Agent Systems (DAMAS)*, July 2005.

[20] Moritz Y. Becker, Cedric Fournet, and Andrew D. Gordon. SecPAL: Design and Semantics of a Decentralized Authorisation Language. Technical report, Microsoft Research, Roger Needham Building 7 J.J. Thompson Avenue, Cambridge, CB3 0FB, UK, September 2006.

[21] D. Bell and L. Lapadula. Secure computer system unified exposition and multics interpretation. Technical Report MTR-2997, MITRE, Bedford, MA, 1975.

[22] F Bellifemine, G Caire, and T Trucco. *Jade Security Guide (Jade 3.3)*. TILAB, February 2005.

[23] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE - A FIPA compliant agent framework. In *PAAM'99, London, April 1999*, pages 97–108, 1999.

[24] Messaoud Benantar. *Access Control Systems, Security, Identity Management and Trust Models*. Springer Science+Business Media Inc., 233 Spring Street, New York, NY 10013, USA, 2006.

[25] Federico Bergenti and Alessandro Ricci. Three Approaches to the Coordination of Multiagent Systems. In *SAC 2002, Madrid, Spain*, 2002.

[26] Shimshon Berkovits, Joshua D. Guttman, and Vipin Swarup. Authentication for Mobile Agents. In *Mobile Agents and Security*, pages 114–136, London, UK, 1998. Springer-Verlag.

[27] Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning. *ACM Transactions on Database Systems*, 23(3):213–285, September 1998.

[28] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security*, 4(3):191–233, 2001.

[29] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur.*, 6(1):71–127, 2003.

[30] K. J. Biba. Integrity considerations for secure computer systems, tr-3153. Technical report, Mitre Cooperation, Bedford, MA, 1977.

[31] Matt Bishop. *Computer Security, Art and Science.* Addison-Wesley, 2003.

[32] N. Borselius. Mobile agent security. *Electronics & Communication Engineering Journal*, 14(5):211–218, October 2002. Available from: `citeseer.nj.nec.com/547211.html`.

[33] Jeffrey M. Bradshaw. An Introduction to Software Agents. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 3–46. AAAI Press / The MIT Press, 1997. Available from: `citeseer.nj.nec.com/bradshaw97introduction.html`.

[34] Jeffrey M. Bradshaw, Hyuckchul Jung, Shri Kulkarni, Matthew Johnson, Paul Feltovich, James Allen, Larry Bunch, Nathanael Chambers, Lucian Galescu, Renia Jeffers, Niranjan Suri, William Taysom, and Andrzej Uszok. Kaa: policy-based explorations of a richer model for adjustable autonomy. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 214–221, New York, NY, USA, 2005. ACM Press.

[35] David Brewer and Michael Nash. The Chinese Wall Policy. In *IEEE Symposium on Research in Security and Privacy*, pages 206–214, May 1989.

[36] Rodney Brooks. New Approaches to Robotics. *Science*, 253:1227–1232, 1991.

[37] Rodney Brooks. *Cambrian Intelligence, the early history of the new AI.* The MIT Press, Cambridge Massachusetts, 1999.

[38] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[39] Seraphin Calo and Jorge Lobo. A Basis for Comparing Characteristics of Policy Systems. In *Seventh IEEE International Workshop on Policies for Distributed Systems and Networks POLICY2006*, pages 183–192, London, Ontario, Canada, June 2006.

[40] A. Cau, B. Moszkowski, and H. Zedan. The ITL homepage. online. Available from: `http://www.cse.dmu.ac.uk/STRL/ITL`.

[41] Antonio Cau and Hussein *Zedan*. *Refining Interval Temporal Logic* Specifications. In M. Bertran and T. Rus, editors, *Transformation-Based Reactive Systems Development*, volume 1231 of *LNCS*, pages 79–94, AMAST, 1997. Springer-Verlag.

[42] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? Technical Report RC 19887 (December 21, 1994 - Declassified March 16, 1995), IBM Research Division, Yorktown Heights, New York, 1994. Available from: `citeseer.ist.psu.edu/chess95mobile.html`.

[43] Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

[44] Cougaar. Cougaar Web Pages. online, 2005. Available from: `http://www.cougaar.org`.

[45] Robert Crook, Darrel Ince, Luncheng Lin, Bashar Nuseibeh, and The Open University. Security Requirements Engineering: When Anti-Requirements Hit the Fan. In *IEEE Joint International Conference on Requirements Engineering (RE'02)*, page 203, September 2002.

[46] N. Damianou, N. Dulay, E. Lupu, and M Sloman. The Ponder Specification Language. In *Workshop on Policies for Distributed Systems and Networks (Policy2001)*, January 2001.

[47] Nicedemos Damianou. *A Policy Framework for Management of Distributed Systems.* PhD thesis, University of London, February 2002.

[48] Nicodemos Damianou, Arosha K Bandra, Morris Sloman, and Emil Lupu. A Survey of Policy Specification Approaches, 2002. Available from: `http://www.doc.ic.ac.uk/~mss/Papers/PolicySurvey.pdf`.

[49] DARPA. DAML: The DARPA Agent Markup Language. online, 2000. Available from: `http://www.daml.org/`.

[50] Mehdi Dastani, Joris Hulstijn, Frank Dignum, and John-Jules Ch. Meyer. Issues in multiagent system development. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 922–929, Washington, DC, USA, 2004. IEEE Computer Society.

[51] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[52] Mark d'Inverno and Michael Luck. *Understanding Agent Systems.* Springer Verlag, 2004.

[53] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for Mobile Agents: Authentication and State Appraisal. In *Proceedings of the Fourth European Symposium on Research in Computer Security*, pages 118–130, Rome, Italy, 1996. Available from: `citeseer.nj.nec.com/farmer96security.html`.

[54] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for Mobile Agents: Issues and Requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 519–597–130, Baltimore, 1996.

[55] Richard Fikes and Nils Nilsson. STRIPS: A new Approach to the application of Theorem Proving to Problem Solving. In *2nd IJCAI, Imperial College, London*, Sept 1971.

[56] FIPA. FIPA Agent Management Specification [SC00023K]. online, 2002. Available from: `http://www.fipa.org/specs/fipa00023/SC00023K.html`.

[57] FIPA. FIPA Communicative Act Library Specification [SC00037J]. online, December 2002. Available from: `http://www.fipa.org/specs/fipa00037/SC00037J.html`.

[58] M. Fisher. A survey of Concurrent METATEM – the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic - Proceedings of the First Intemational Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Heidelberg, Germany, 1994.

[59] Gleb Frank, Jessica Jenkins, and Richard Fikes. JTP: An Object-Oriented Modular Reasoning System, 2005. Available from: `http://ksl.stanford.edu/software/jtp/`.

[60] Stan Franklin and Art Graesser. Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents. In *Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, pages 21–35. Springer-Verlag, 1996. Available from: `http://www.cs.memphis.edu/~franklin/AgentProg.html`.

[61] Pedro Gamma, Carlos Ribero, and Paulo Ferreira. A Scalable History-based Policy Engine. In *Proceedings of the 7th International Workshop on Policies for Distributed Systems and Networks*, 2006.

[62] Rodolfo Gomez and Howard Bowman. PITL2MONA: Implementing a Decision Procedure for Propositional Interval Temporal Logic. *Journal of Applied Non-Classical Logics*, 14(1-2):105–148, unknown 2004. Issue on Interval Temporal Logics and Duration Calculi. V. Goranko and A. Montanari guest eds. Available from: `http://www.cs.kent.ac.uk/pubs/2004/1891`.

[63] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, Reading, MA, 1996.

[64] Tyrone Grandison and Morris Sloman. A Survey of Trust in Internet Applications. *IEEE Communications Surveys and Tutorials*, 3(4), September 2000. http://www.comsoc.org/livepubs/surveys/public/2000/dec/index.html. Available from: `http://pubs.doc.ic.ac.uk/TrustSurvey/`.

[65] David Gries. *The Science of Programming.* Springer Verlag, New York, 1985.

[66] Roger William Stephen Hale. *Programming in Temporal Logic.* PhD thesis, Trinity College, University of Cambridge, October 1988.

[67] J. Halpern and V. Weissman. Using first-order logic to reason about policies. In *Proceedings of the Computer Security Foundations Workshop (CSFW'03)*, 2003. Available from: `citeseer.ist.psu.edu/halpern03using.html`.

[68] James Hoagland. *Specifying and Implementing Security Policies Using LaSCO, the Language for Security Constraints on Objects.* PhD thesis, University of California, 2000.

[69] IBM Cooperation. Enterprise Privacy Authorisation Language (EPAL) Version 1.2. submitted to the W3C, 2003. Available from: `http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/`.

[70] ISO/IEC. ISO/IEC 14977:1996 International standard EBNF notation. online, 1996. Available from: `http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`.

[71] ISO/IEC. ISO/IEC 10181-3:1996 Information technology – Open Systems Interconnection – Security frameworks for open systems: Access control framework, March 2006. Available from: `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=18199`.

[72] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[73] Sushil Jajodia, Pierangela Samarati, V. S. Subrahmanian, and Eliza Bertino. A unified framework for enforcing multiple access control policies. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 474–485, New York, NY, USA, 1997. ACM Press.

[74] Helge Janicke, Antonio Cau, François Siewe, Hussein Zedan, and Kevin Jones. A Compositional Event & Time-based Policy Model. In *Procceedings of POLICY2006, London, Ontario, Canada.* IEEE, 2006.

[75] Helge Janicke, François Siewe, Kevin Jones, Antonio Cau, and Hussein Zedan. Analysis and Run-time Verification of Dynamic Security Policies. In Robert Ghanea-Hercock and Mark Greaves and Nick Jennings and Simon Thompson, editor, *In Proceedings of The First Workshop on Defence Applications for Multi-Agent Systems (DAMAS'05), at AAMAS'05, Utrecht, Netherlands.*, 2005.

[76] W. Jansen. Countermeasures for Mobile Agent Security. *Computer Communications, Special Issue on Advances in Research and Application of Network Security*, November 2000. Available from: `citeseer.ist.psu.edu/jansen00countermeasures.html`.

[77] N. R. Jennings, K. Sycara, and M. Wooldridge. A Roadmap of Agent Research and Development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998. Available from: `citeseer.nj.nec.com/jennings98roadmap.html`.

[78] Jan Jrjens. Towards Development of Secure Systems using UML. Technical Report PRG-TR-09-00, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK, 2000.

[79] Jan Jrjens. UMLsec: Extending UML for Security Systems Development. In *UML 2002*, LNCS, Dresden, Sep. 30 - Oct. 4 2002. Springer-Verlag.

[80] Neeran Karnik and Anand Tripathi. Security in the Ajanta Mobile Agent Programming System. *Software Practice and Experience*, pages 301–329, January 2001.

[81] Alan Kay. Computer Software. *Scientific American*, 251(3):53–59, 1984.

[82] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[83] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. An Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[84] Butler Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium of Information Science and Systems*, pages 437–443, March 1971.

[85] Carl E. Landwehr. A Survey of Formal Models for Computer Security. Technical report, Naval Research Laboratory, 1981.

[86] Leonard J. LaPadula and D. Elliot Bell. Secure Computer Systems: Mathematical Foundations. Technical report, MITRE Technical Report 2547, 1973.

[87] Nicholas Lhuillier. Security in Multi-Agent *Systems: JADE-S goes Distributed.* TelcomLab Italia - exp in search of innovation, Volume 3 - n. 3 - September 2003. Special issue on Jade., September 2003.

[88] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 2005.

[89] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, 1977.

[90] Michael Luck, Ronald Ashri, and Mark d'Inverno. *Agent-Based Software Development.* Artech House Publishers, 2004.

[91] Pettie Maes. Agents that Reduce Work and Information Overload. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 145–164. AAAI Press / The MIT Press, 1997.

[92] Viviana Mascardi, Daniela Demergasso, and Davide Ancona. Languages for Programming BDI-style Agents: an Overview. In Flavio Corradini, Flavio De Paoli, Emanuela Merelli, and Andrea Omicini, editors, *WOA*, pages 9–15. Pitagora Editrice Bologna, 2005.

[93] Sun Microsystems. Java 2 SDK Homepage, June 2005. Available from: `http://java.sun.com/products/archive/j2se/1.4.0_02/`.

[94] Luc Moreau, Jeff Bradshaw, Maggie Breedy, Larry Bunch, Pat Hayes Matt Johnson, Shri Kulkarni, James Lott, Niranjan Suri, and Andrzej Uszok. Behavioural Specification of Grid Services with the KAoS Policy Language. In *Proceedings 2005 IEEE International Symposium on Cluster Computing and the Grid*, 2005.

[95] Antonio Moreno, Aida Calla, and Alexandre Viejo. Using JADE-LEAP to implement agents in mobile devices. TelcomLab Italia - exp in search of innovation, Volume 3 - n. 3 - September 2003. Special issue on Jade., September 2003.

[96] Carol Morgan. *Programming from Specifications.* Prentice Hall International (UK) Ltd., second edition, 1998. Available from: `http://users.comlab.ox.ac.uk/carroll.morgan/PfS/`.

[97] Till Mossakowski, Michael Drouineaud, and Karsten Sohr. A temporal-logic extension of role-based access control covering dynamic separation of duties. In *time-ictl, 10th International Symposium on Temporal Representation and Reasoning and Fourth International Conference on Temporal Logic*, volume 1, page 83, 2003.

[98] B. Moszkowski. Some very compositional temporal properties. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, volume A-56 of *IFIP Transactions*, pages 307–326. IFIP, Elsevier Science B.V. (North–Holland), 1994.

[99] Ben Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, England, 1986. Available from: `http://www.cse.dmu.ac.uk/STRL/ITL/itlhomepagese7.html#x8-100007`.

[100] Ben Moszkowski. Compositional Reasoning about Projected and Infinite Time. In *Proceedings of the 1st IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95)*, 1995.

[101] Ben Moszkowski. Compositional Reasoning using Interval Temporal Logic and tempura. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1486 of *LNCS*, pages 439–464, Berlin, 1998. Springer Verlag.

[102] Syed Naqvi, Phillipe Massonet, and Alvaro Arenas. A study of languages for the specification of grid security policies. Technical Report TR-0037, CoreGRID - Network of Excellence, April 2006.

[103] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997. Available from: `citeseer.ist.psu.edu/50371.html`.

[104] George C. Necula and Peter Lee. Safe, Untrusted Agents Using Proof-Carrying Code. *Mobile Agents and Security, Springer-Verlag, Berlin 1998*, 1419:61–91, 1998.

[105] R. Needham and M. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[106] Nicholas Negroponte. Agents: From Direct Manipulation to Delegation. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 57–66. AAAI Press / The MIT Press, 1997.

[107] Hyacinth S. Nwana. Software Agents: An Overview. *Knowledge Engineering Review*, 11(3):205–244, October/November 1995. Available from: `citeseer.nj.nec.com/nwana96software.html`.

[108] OASIS. eXtensible Access Control Markup Language (XACML) Version 2.0, February 2005. Available from: `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml#XACML20`.

[109] Andrea Omicini. Towards a Notion of Agent Coordination Context. In Dan C. Marinescu and Craig Lee, editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187–200. CRC Press, October 2002.

[110] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Formal Specification and Enactment of Security Policies through Agent Coordination Contexts. *Electronic Notes in Theoretical Computer Science*, 85(3), 2003. Available from: `http://www1.elsevier.com/gej-ng/31/29/23/138/23/25/85.3.003.pdf`.

[111] R. Pandey and B. Hashii. Providing Fine-Grained Access Control For Java Programs Through Binary Editing. In *Concurrency: Practice and Experience*, volume 12, pages 1405–1430, 2000.

[112] Jaehong Park and Ravi Sandhu. Towards usage control models: beyond traditional access control. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 57–64, New York, NY, USA, 2002. ACM Press.

[113] Jaehong Park, Xinwen Zhang, and Ravi S. Sandhu. Attribute mutability in usage control. In Csilla Farkas and Pierangela Samarati, editors, *DBSec*, pages 15–29. Kluwer, 2004.

[114] Alexander Pretschner, Manuel Hilty, and David Basin. Distributed usage control. *Commun. ACM*, 49(9):39–44, 2006. Available from: `http://doi.acm.org/10.1145/1151030.1151053`.

[115] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995. Available from: `citeseer.nj.nec.com/rao95bdi.html`.

[116] Anand S. Rao and Michael P. Georgeff. Modeling Rational Agents within a BDI-Architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991. Available from: `citeseer.ist.psu.edu/rao91modeling.html`.

[117] Carlos N. Ribeiro, Andre Zuquete, Paulo Ferreira, and Paulo Guedes. Enforcing Obligations with Security Monitors. In *The Third International Conference on Information and Communication Security (ICICS'2001)*, November 2001.

[118] Carlos N. Ribeiro, Andre Zuquete, Paulo Ferreira, and Paulo Guedes. SPL: An access control language for security policies with complex constraints. In *Network and Distributed System Security Symposium (NDSS'01)*, February 2001.

[119] P. Samarati and S. Vimercati. Access Control: Policies, Models, and Mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design (Tutorial Lectures)*, pages 137–196. Springer-Verlag, September 2000.

[120] Ravi Sandhu and Jaehong Park. The UCON_{ABC} usage control model. In *Proceeding of the Second International Workshop on Mathematical Method, Models and Architectures for Computer Networks Security*, 2003.

[121] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[122] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[123] Yoav Shoham. An Overview of Agent-Oriented Programming. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 271–290. AAAI Press / The MIT Press, 1997.

[124] François Siewe. *A Compositional Framework for the Development of Secure Access Control Systems.* PhD thesis, Software Technology Research Laboratory, Department of Computer Science and Engineering, De Montfort University, Leicester, 2005.

[125] Guttorm Sindre and Andreas L. Opdahl. Capturing Security Requirements by Misuse Cases. In *14th Norwegian Informatics Conference (NIK'2001)*, Troms, Norway, November 26 – 28 2001.

[126] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2:333–360, 1994. Available from: `citeseer.ist.psu.edu/sloman94policy.html`.

[127] David Canfield Smith, Allen Cypher, and Jim Spohrer. KidSim: Programming Agents without a Programming Language. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 167–190. AAAI Press / The MIT Press, 1997.

[128] Roshan K. Thomas. Team-based access control (TMAC): a primitive for applying role-based access controls in collaborative environments. In *RBAC '97: Proceedings of the second ACM workshop on Role-based access control*, pages 13–19, New York, NY, USA, 1997. ACM Press.

[129] TILAB. Java Agent DEvelopment Framework (JADE). online, 2005. Available from: `http://jade.tilab.com/`.

[130] Gianluca Tonti, Jeffery M. Bradshaw, Renia Jeffers, Rebecca Montanari, Niranjan Suri, and Andrezej Uszok. Semantic Web Languages for Policy Representation and Reasoning: A Comparison of KAoS, Rei, and Ponder. In *The SemanticWeb - ISWC-2003*, volume 2870/2003 of *Lecture Notes in Computer Science*, pages 419–437. Springer Berlin / Heidelberg, 2003.

[131] Tripathi and Karnik. Ajanta Homepage, 2005. Available from: `http://www.cs.umn.edu/Ajanta`.

[132] Anand Tripathi, Tanvir Ahmed, and Richa Kumar. Specification of Secure Distributed Collaboration Systems. Technical report, April 2003. Available at http://www.cs.umn.edu/Ajanta. Available from: `http://www.cs.umn.edu/Ajanta`.

[133] Anand R. Tripathi, Neeran Karnik, Tanvir Ahmed, and Ram D. Singh. Design of the Ajanta System for Mobile Agent Programming. *Journal of Systems and Software*, 62(2):123–140, May 2002.

[134] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. KAoS policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement. In *Proceedings POLICY 2003 Policies for Distributed Systems and Networks*, pages 93–96, June 2003.

[135] W3C. Owl: Web ontology language, February 2004. Available from: `http://www.w3.org/TR/owl-features/`.

[136] Scott A. Wallace. S-Assess: A Library for Behavioural Self-Assesment. In *Proceedings 4th International Joint Conference on Autonomous Agents & Multi Agent Systems, AAMAS05*, volume 1, pages 256–263, 2005.

[137] Webster. Webster Dictionary, 1913. Available from: `http://machaut.uchicago.edu/cgi-bin/WEBSTER.sh?WORD=agent`.

[138] Duminda Wijesekera and Sushil Jajodia. A Propositional Policy Algebra for Access Control. *ACM Transactions on Information and Systems Security*, 6(2):286–325, May 2003.

[139] Michael Wooldridge. *Multi Agent Systems.* John Wiley & Sons Ltd, 2002.

[140] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995. Available from: `citeseer.ist.psu.edu/article/wooldridge95intelligent.html`.

[141] Michael J. Wooldridge. *The Logical Modelling of Comutational Multi-Agent Systems.* PhD thesis, University of Manchester, 1992.

[142] Franco Zambonelli, Nicholas Jennings, and Michael Wooldridge. Developing Multiagent Systems: The Gaia Methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, July 2003. Available from: `http://www.ecs.soton.ac.uk/~nrj/download-files/tosem03.pdf`.

[143] Hussein Zedan, Antonio Cau, Zhiqiang Chen, and Hongji Yang. ATOM: An Object-based Formal Method for Real-Time Systems. *Annals of Software Engineering*, 7:235–256, 1999. Available from: `citeseer.ist.psu.edu/zedan99atom.html`.

[144] Xinwen Zhang, Jaehong Park, Francesco Parisi-Presicce, and Ravi Sandhu. A logical specification for usage control. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 1–10, New York, NY, USA, 2004. ACM Press.

[145] Xinwen Zhang, Francesco Rarisi-Presicce, Jaehong Park, and Ravi Sandhu. Formal Model and Policy Specification of Usage Control. *ACM Transactions on Information and System Security*, 2005.

Glossary

A

access control The function of access-control is to control which principals [subjects] have access to which resources [objects] in the system — which files they can read, which programs they can execute, how they share data with other principals, and so on. [8].

action In the contect of Policies: An action is performed by a subject on an object. An example are the `-rwx` actions that are protected in a UNIX file system. actions can denote the mode of access to the object or represent a more complex access, e.g. method invocations.

Actor An Actor is a self contained, interactive and continuously executing object.

Agent Communication Language (ACL) Agents in open environments generally communicate using an Agent Communication Language. This language usually provides performatives to distinguish the type of the message and beside a message-content, other mandatory parameters that are needed for message passing.

Agent Orientated Programming (AOP) Programming paradigm in that the basic structure of definiton is an agent. Agents, unlike objects in Object Oriented Programming do not only encapsulate methods and attributes, but also the control of these. This allows to specify a system that is build out of highly modular and independent software entities (agents) that exibit control over their actions and state.

Artificial Intelligence (AI) Artificial Intelligence is a branch of computer science that deals with intelligent behavior, learning, and adaptation in machines. Research in AI is concerned with producing machines to automate tasks requiring intelligent behaviour. [Source: Wikipedia, page Artificial Intelligence as of 23rd October 2006].

Artificial Intelligence Planning (AIP) Subfield of Artificial Intelligence that concerns itself with knowing *what to do*. It is usually associated with the Stanford Research Institute Problem Solver planning system.

authentication Authentication is the process of attempting to verify the digital identity of the sender of a communication such as a request to log in. The sender

being authenticated may be a person using a computer, a computer itself or a computer program.

authorisation Authorisation determines whether access to system resource is allowed or denied, based on a subject's identity or credentials.

Authorisation Specification Language (ASL) A policy model/ language that was developed by Jajodia et.al. [72].

B

Belief, Desire, Intention (BDI) *Arguably, the most successful agent architectures are founded on the BDI model, in which agents continually monitor their environments and act to change them, based on three mental attitudes of belief, desire, and intention, representing informational, motivational and decision-making capabilities. Architectures based on the BDI model represent beliefs, desires and intentions as data structures, which determine the operation of an agent*[90].

D

delegation Delegation policies define the conditions under which a subject can delegate a specific access right (i.e. the right to perform an action on an object) to another subject.

Deliberation-Enforcement-Execution (DEE) The main phases in the execution model of a single SANTA agent.

Discretionary Access Control (DAC) In Discretionary Access Control, the system's security policy is under the discretion of individuals, e.g. the owner of a resource. A typical example is the file permission in a UNIX file system.

Distributed Artificial Intelligence (DAI) Subfield of Artificial Intelligence that can be further divided in to main camps Distributed Problem Solving and Multi Agent System.

Distributed Problem Solving (DPS) Subfield of Distributed Artificial Intelligence that, according to Jennings et.al. *considers how a particular problem can be solved by a number of modules, which cooperate in dividing and sharing knowledge about the problem and its evolving solution.*

E

Enterprise Privacy Authorisation Language (EPAL) *EPAL is a formal language for writing enterprise privacy policies to govern data handling practices in IT systems according to fine-grained positive and negative authorization rights. It concentrates on the core privacy authorization while abstracting data models and*

user-authentication from all deployment details such as data model or user-authentication. [taken from http://www.w3.org/Submission/EPAL/. Formal in this context does not mean mathematical.

eXtended Access Control Mark-up Language (XACML) Extensible Access Control Markup Language, or XACML, was approved and became an OASIS standard in February 2003. XACML defines a general policy language used to protect resources as well as an access decision language. [taken from http://dev2dev.bea.com/pub/a/2004/02/xacml.html].

eXtensible Markup Language (XML) The Extensible Markup Language (XML) is a W3C-recommended general-purpose markup language for creating special-purpose markup languages, capable of describing many different kinds of data. [Wikipedia page Extensible Markup Language].

I

integrity Integrity policies define constraints on the execution of an action on an object by a specific subject.

Interval Temporal Logic (ITL) ITL is a linear, interval-based, first-order temporal logic. It allows to express the behaviour of a system over a sequence of states. The ITL homepage [40] provides a complete introduction.
See also Tempura.

M

Mandatory Access Control (MAC) In Mandatory Access Control the system's security policy is under the control of a dedicated administrator. Typically used for military or governmental security policies, where subjects are associated with a clearance label and objects with a security label.

Massachusetts Institute of Technology (MIT) The Massachusetts Institute of Technology is based in Boston, USA. See http://web.mit.edu.

Multi Agent System (MAS) Originally a research area of Distributed Artificial Intelligence. According to Jennings et.al. [77] the term MAS is now used *"to refer to a system, that is composed of multiple (semi-) autonomous components"*.

N

National Institute of Standards and Technology (NIST) The National Institute of Standards and Technology (NIST, formerly known as The National Bureau of Standards) is a non-regulatory agency of the United States Department of Commerce's Technology Administration. The institute's mission is to promote U.S. innovation and industrial competitiveness by advancing measurement science,

standards, and technology in ways that enhance economic security and improve quality of life. [Source: Wikipedia, page National Institute of Standards and Technologies as of 23rd October 2006].

Non-Discretionary Access Control (NDAC) Combines MAC and DAC by leaving some discretion to the user, however centrally managing aspects like the delegation of rights.

O

object In the context of Policies: An object is the target of an action execution by a subject.

Object Oriented Programming (OOP) Programming paradigm, in that the basic structure of definition is an object, that represents an instantiation of a class. Objects encapsulate attributes and methods.
See also Agent Orientated Programming.

obligation Obligation policies define the conditions or events (sometimes also called triggers) under which a subject has to perform a specific action on an object.

Observe, Orientate, Decide, Act (OODA) The OODA Loop is a concept originated by military strategist Col. John Boyd of the United States Air Force. Its main outline consists of four overlapping and interacting processes: Observe, Orient, Decide and Act. [Wikipedia page OODA Loop].

Organisation for the Advancement of Structured Information Standards (OASIS) Organization for the Advancement of Structured Information Standards, a global consortium that develops data representation standards for use in computer software. [Source: Wikipedia, OASIS (organization) as of 23rd October 2006].

P

Policy Decision Point (PDP) A logical entity that makes policy decisions for itself or for other network elements that request such decisions. [Source RFC3198, Network Working Group, The Internet Society (2001)].

Policy Enforcement Point (PEP) A logical entity that enforces policy decisions. [Source RFC3198, Network Working Group, The Internet Society (2001)].

Public-Private Key (PPK) Public key cryptography is a form of cryptography which generally allows users to communicate securely without having prior access to a shared secret key. This is done by using a pair of cryptographic keys, designated as public key and private key. [Source: Wikipedia, page Public-key cryptography as of 23rd October 2006].

R

Role-Based Access Control (RBAC) In Role-Based Access Control the granting of access rights is abstracted from the concrete user to roles, in that the user can act. This is beneficial, since role-hierarchies reflect the organisations structure and are therefore less likely to change. This is especially true for companies with a high staff-turnover. The advantage is that the security policy does not need to change, only the assignment of user to roles. This reduces security administration cost.

S

SANTA Wide-Spectrum Language (SANTA-WSL) Linguistic support for the specification of Secure Multi Agent System within the SANTA framework. Wide-spectrum means that the language accomodates both, specification constructs and concrete implementation.

Secure Multi Agent System (SMAS) A SMAS consists of reactive agents, objects, security policies and enforcement mechanisms.

security policy Policies are rules governing the choices in the behaviour of a system. They allow the separation of rules that govern choices in the behaviour of the system, from the system's functionality. [16].

Security Policy Analysis Tool (SPAT) SPAT is a tool-kit for the early prototyping and analysis of dynamically changing security policies. SPAT uses Tempura to simulate environment and policy behaviour.

Stanford Research Institute Problem Solver (STRIPS) STRIPS is a planning system consisting of three components:

1. symbolic model of the agent's environment, typically in a subset of first-order predicate logic.
2. symbolic specification of the agents actions.
3. planning algorithm that uses 1., 2. and a representation of the agents goal state to create a plan, which specifies how the agent can act to achieve the goal.

.

subject In the context of Policies: A subject is an entity that participates in the execution of the system. It can be for example a user, an agent/process executing on behalf of a user, groups, roles etc.

T

Team-based Access Control (TMAC) Team-based Access Control is an approach to apply Role-Based Access Control in collaborative environments. A *team* is an

abstraction that encapsulates a collection of users in specific roles with the objective of accomplishing a task. [128].

Temporal Role-based Access Control (TRBAC) An extension to RBAC that includes support for the definition of time and periodicity constraints. See [28].

Tempura Executable subset of Interval Temporal Logic, see [99].

U

Usage Control (UCON) Usage Control (UCON) encompasses traditional access control, trust management, and digital rights management and goes beyond them in its definition and scope. It has been developed by Sandhu, Park and Zhang e.g. [112].

W

World Wide Web (WWW) The World Wide Web ("WWW" or simply the "Web") is a global, read-write information space. Text documents, images, multimedia and many other items of information, referred to as resources, are identified by short, unique, global identifiers called Uniform Resource Identifiers (URIs) so that each can be found, accessed and cross referenced in the simplest possible way. [Wikipedia page World Wide Web].